Oracle® Retail Integration Bus

Service-Oriented Architecture Enabler Tool Guide

Release 14.0

E49440-01

December 2013

ORACLE®

Oracle Retail Service-Oriented Architecture Enabler Tool Guide, Release 14.0

E49440-01

Primary Author: Sanal Parameswaran

Contributing Author: Gloreen Soans

reconfigurations, reassembly or reverse assembly, re-engineering or reverse engineering and recompilations or reverse compilations of the VAR Applications or any derivatives of the VAR Applications. You acknowledge that it shall be a breach of the agreement to utilize the relationship, and/or confidential information of the VAR Applications for purposes of competitive discovery.

The VAR Applications contain trade secrets of Oracle and Oracle's licensors and Customer shall not attempt, cause, or permit the alteration, decompilation, reverse engineering, disassembly or other reduction of the VAR Applications to a human perceivable form. Oracle reserves the right to replace, with functional equivalent software, any of the VAR Applications in future releases of the applicable program.

Contents

9 Implementation Guidelines

10 Implementation Guidelines For Restful web services

11 Web Services Security Setup Guidelines

A Appendix: Installer Screens

Send Us Your Comments

Oracle Retail Service-Oriented Architecture Enabler Tool Guide, Release 14.0

Oracle welcomes customers' comments and suggestions on the quality and usefulness of this document.

Your feedback is important, and helps us to best meet your needs as a user of our products. For example:

- Are the implementation steps correct and complete?
- Did you understand the context of the procedures?
- Did you find any errors in the information?
- Does the structure of the information help you with your tasks?
- Do you need different information or graphics? If so, where, and in what format?
- Are the examples correct? Do you need more examples?

If you find any errors or have any other suggestions for improvement, then please tell us your name, the name of the company who has licensed our products, the title and part number of the documentation and the chapter, section, and page number (if available).

> **Note:** Before sending us your comments, you might like to check that you have the latest version of the document and if any concerns are already addressed. To do this, access the new Applications Release Online Documentation CD available on My Oracle Support and www.oracle.com. It contains the most current Documentation Library plus all documents revised or released recently.

Send your comments to us using the electronic mail address: retail-doc_us@oracle.com

Please give your name, address, electronic mail address, and telephone number (optional).

If you need assistance with Oracle software, then please contact your support representative or Oracle Support Services.

If you require training or instruction in using Oracle software, then please contact your Oracle local office and inquire about our Oracle University offerings. A list of Oracle offices is available on our Web site at www.oracle.com.

Preface

The *Oracle Retail Service-Oriented Architecture Enabler (RSE) Tool Guide* provides information about the tool as well as installation instructions.

Audience

The *Oracle Retail Service-Oriented Architecture Enabler (RSE) Tool Guide* is written for the following audience:

- Database administrators (DBA)
- System analysts and designers
- Integrators and implementation staff

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For more information, see the following documents in the Oracle Retail Integration Bus 14.0 documentation set:

- *Oracle Retail Integration Bus Implementation Guide*
- *Oracle Retail Integration Bus Security Guide*
- *Oracle Retail Integration Bus Installation Guide*
- *Oracle Retail Integration Bus Release Notes*
- *Oracle Retail Integration Bus Hospital Administration Guide*
- *Oracle Retail Integration Bus Operations Guide*
- *Oracle Retail Integration Bus Support Tools Guide*

- *Oracle Retail Enterprise Integration Guide*
- *Oracle Retail Functional Artifacts Guide*
- *Oracle Retail Integration Bus Integration Gateway Services Guide*
- *Oracle Retail Functional Artifact Generator Guide*
- *Oracle Retail Integration Bus Data Model*
- *Oracle Retail PL-SQL Payload Mapper Guide (ID 1590674.1)*

Customer Support

To contact Oracle Customer Support, access My Oracle Support at the following URL:

https://support.oracle.com

When contacting Customer Support, please provide the following:

- Product version and program/module name
- Functional and technical description of the problem (include business impact)
- Detailed step-by-step instructions to re-create
- Exact error message received
- Screen shots of each step you take

Review Patch Documentation

When you install the application for the first time, you install either a base release (for example, 13.2) or a later patch release (for example, 13.2.6). If you are installing the base release, additional patch, and bundled hot fix releases, read the documentation for all releases that have occurred since the base release before you begin installation. Documentation for patch and bundled hot fix releases can contain critical information related to the base release, as well as information about code changes since the base release.

Improved Process for Oracle Retail Documentation Corrections

To more quickly address critical corrections to Oracle Retail documentation content, Oracle Retail documentation may be republished whenever a critical correction is needed. For critical corrections, the republication of an Oracle Retail document may at times not be attached to a numbered software release; instead, the Oracle Retail document will simply be replaced on the Oracle Technology Network Web site, or, in the case of Data Models, to the applicable My Oracle Support Documentation container where they reside.

This process will prevent delays in making critical corrections available to customers. For the customer, it means that before you begin installation, you must verify that you have the most recent version of the Oracle Retail documentation set. Oracle Retail documentation is available on the Oracle Technology Network at the following URL:

http://www.oracle.com/technetwork/documentation/oracle-retail-100266.html

An updated version of the applicable Oracle Retail document is indicated by Oracle part number, as well as print date (month and year). An updated version uses the

same part number, with a higher-numbered suffix. For example, part number E123456-02 is an updated version of a document with part number E123456-01.

If a more recent version of a document is available, that version supersedes all previous versions.

Oracle Retail Documentation on the Oracle Technology Network

Documentation is packaged with each Oracle Retail product release. Oracle Retail product documentation is also available on the following Web site:

`http://www.oracle.com/technology/documentation/oracle_retail.html`

(Data Model documents are not available through Oracle Technology Network. These documents are packaged with released code, or you can obtain them through My Oracle Support.)

Documentation should be available on this Web site within a month after a product release.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1
Introduction

The purpose of the Retail Service-Oriented Architecture Enabler (RSE) tool is to provide a standard, consistent way to develop Web services for PL/SQL and Java EE applications. Because it allows them to expose their business logic, the focus of development can be on the business logic code, not on the Web service infrastructure.

The RSE tool creates Web service provider end-points, consumer clients for Web service providers, and templates for interfacing with PL/SQL APIs and Java EE APIs.

The tool also produces design time and run time artifacts. It works in conjunction with another RTG tool, the Retail Functional Artifact Generator.

Note: For more information on the tool, see the *Oracle Retail Functional Artifact Generator Guide*.

Major Features of the RSE Tool

The following is a list of the essential features of the RSE tool:

- The RSE tool is standards based.
- The RSE tool supports SOAP and RESTful based web services.
- SOAP based web services:
  - All services are generated in a consistent and standard manner.
  - All services are SOAP/HTTP based.
  - All services comply with the JAX-WS specification.
  - All services are WS-Addressing enabled.
  - WS-Security can be plugged into these Web services without any code change.
  - All Web services are Document Literal Wrapped.
  - Generated services are capable of using SOAP headers.
- RESTful based web services:
  - REST is an architectural style, not a standard.
  - The services comply with the JAX-RS specification.
  - The services support all HTTP methods.
  - WS-Security can be plugged into these Web services without any code change.
- The RSE tool generates technology-specific API templates for PL/SQL APIs and Java EE.

  - It supports PL/SQL as a Web service provider.
  - PL/SQL code can directly call any third party SOAP/HTTP based Web services.
  - It supports java code as a Web service provider.
  - It supports java code as a Web service consumer.
- Generation by the RSE tool is controlled by a single Service Definition Library XML file.
  - By creating Web services from the high level abstraction in the Service Definition Library, top down Web services development is supported.
  - All service operation inputs and outputs are validated against the XML schema.
  - There is a single source of truth for all service and domain object documentation.
  - The same documentation is propagated to static WSDL, Java/PLSQL API code, UDDI published content, and live WSDL.
  - The Service Definition Library XML file is a service-oriented architecture governance asset.
- The generated services deploy in any Java EE 5 with JDK 7 compliant application server, with certification on Oracle WebLogic Server. (Services are deployable to a clustered Java EE application server.)
- The generated services are callable as SOAP based Web services over SOAP/HTTP, local EJB calls, remote EJB calls, or POJO services.
- All services support Web service versioning strategy.
- All generated Web services are forward and backward compatible.
- For every Web service, a static WSDL is generated. (The generated static WSDL pulls in all of the Business Object (BO) and Web service level documentation.)
- All deployed services can be published to any standard UDDI registry.
  - UDDI publishing has been tested with both WebLogicServer and Oracle Service Repository (OSR).
  - Every generated <*appname*>-service.ear contains an Infrastructure Management Service that can "talk to" the UDDI registry and publish all the services available within the .ear to the registry.
- Services can take advantage of Oracle Database Real Application Cluster (RAC).
- The RSE tool has the following built-in functionality:
  - Every service generated has a ping operation to test for network connectivity.
  - A Service Operation Context is passed to both Java EE and PL/SQL service provider API implementation code.
  - The Web service consumer generated has client side asynchronous service invocation capability.
  - User-defined Web service Faults are automatically generated and handled by the infrastructure at runtime. The definitions are made in the Service Definition Library XML file.

- All Web service operations are transactional. A SOAP Fault response automatically rolls back the service operations transaction. A success response automatically commits the service operations transaction.
- Web service consumers do not participate in the Web service provider side transaction. There is no transaction context propagation from client to server.

Concepts

Service-oriented architecture (SOA) is a strategy for constructing business-focused software systems from loosely coupled, interoperable building blocks (called Services) that can be combined and reused quickly, within and between enterprises, to meet business needs (as described in Oracle Fusion Reference Architecture, SOA Foundation Release 1.0).

Service Infrastructure products focus on enabling SOA projects, rather than developing new business function, or providing for other business driven needs. The goal of Service Infrastructure is to enable delivery teams to deliver SOA projects faster, and to make the overall SOA undertaking much more manageable.

The Retail Service-Oriented Architecture Enabler Tool (RSE) is designed and developed to support the creation of Web services by allowing a high level abstraction, higher than the WSDL, and tailored to the business analyst/functional analyst. The Business Analyst can easily understand, define, and design without knowing the intricacies of WSDLs and the technical details of the implementation. This approach is also called top-down Web services development.

What is a Service?

A service can be described as a way of packaging reusable software building blocks to provide functionality to users and to other services. A service is an independent, self-sufficient, functional unit of work that is discoverable, manageable, and measurable, has the ability to be versioned, and offers functionality that is required by a set of users or consumers.

A logical definition of a service has three components:

- Contract: A description of what the service provides (and its constraints).
- Interface: The means by which the service is invoked.
- Implementation: The deployed code and configuration of infrastructure.

Oracle Fusion Reference Architecture (OFRA)

It is important to understand the position and role of the RSE tool within the broader context of service-oriented architecture and development. It is beyond the scope of this document to cover the range of SOA approaches and methodologies, but it is necessary to cover some aspects to place the tool in the appropriate context.

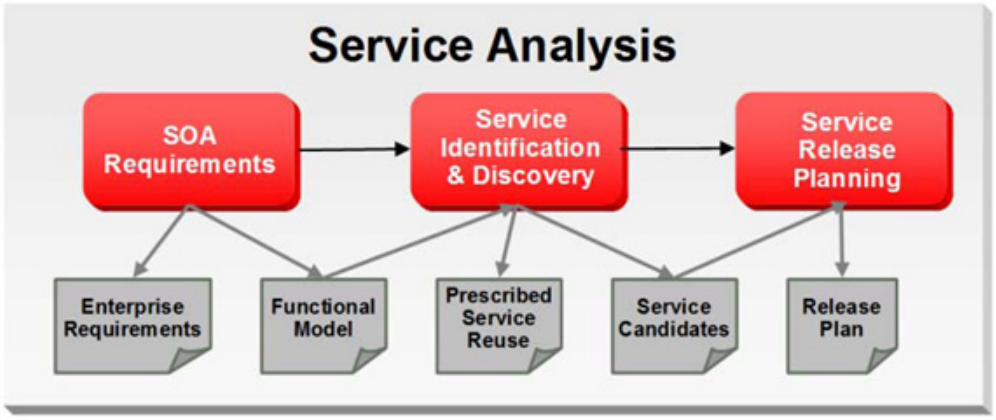
Oracle has developed and published the Oracle Fusion Reference Architecture (OFRA) for building and integrating enterprise-class solutions, part of the IT Strategies from Oracle collection.

The Oracle Fusion Architecture Framework is a collection of assets designed to provide guidance on building solutions for the Oracle Fusion solution environment, which includes the Oracle Fusion Reference Architecture (OFRA). The following diagrams and definitions are from OFRA documentation.

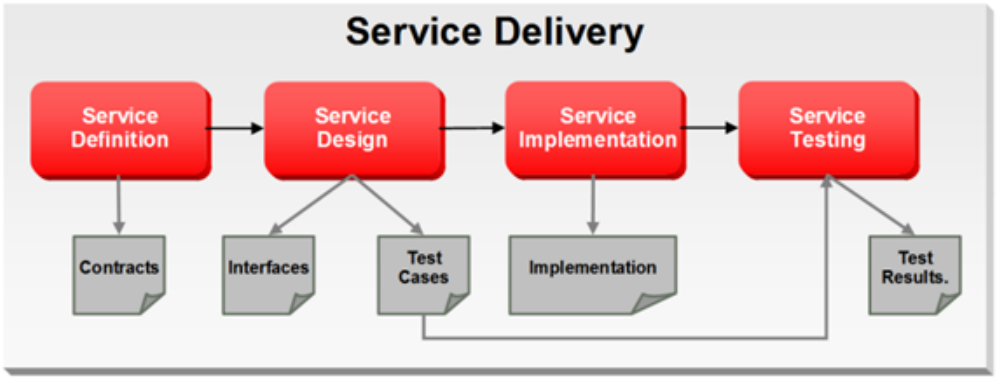
Note: See Oracle Practitioner Guide Software Engineering in an SOA Environment Release 1.0 E14486-01.

The service analysis phase of the Oracle Service Engineering Framework consists of three main sets of engineering practices: SOA Requirements Management, Service Identification & Discovery, and Service Release Planning.

As with traditional software engineering, service engineering also begins with requirements and analysis, as illustrated below:



After Service Analysis, the next phase is Service Delivery, which includes the core delivery engineering activities. In this phase, a service candidate is molded into one or more services. Service candidates entering this phase have been justified for realization and scheduled for release.



Service Delivery begins with Service Definition, which primarily determines service boundaries as well as the construction of the service contract.

Service Design then acts upon the Service contracts to develop the Services' interfaces. The process of defining a Service interface is much more involved than simply coming up with the input and output for the Service. Service design analyzes the contract from the consumer's perspective, and is influenced by factors such as scope (enterprise, LOB, application, and so on), message exchange patterns (MEPs) as well as non-functional requirements such as expected volume, and response time requirements (specified in the contract).

Service Implementation ensures that all aspects of the service contracts are implemented and upheld through the delivery of business logic as well as the deployment to Service Infrastructure. The implementation must faithfully realize the Service Contract and interface which are defined through Service definition and design.

Note: See: Oracle Fusion Reference Architecture, Overview. Release 1.0 E14482-01

Where Does RSE Fit?

The Retail Service-Oriented Architecture Enabler (RSE) is a Service Infrastructure tool developed by Oracle Retail to enable the adoption of service-oriented architecture (SOA) and avoid some of the typical pitfalls of many SOA projects. It addresses many common issues, such as versioning, contract design, security, consistency, reuse, documentation, governance, compliance, and customization. It does this by enforcing SOA Best Practices and patterns that are proven and time tested by various SOA pioneers.

The tool provides the capability for business analysts and developers to define the correct service contract. It provides ease-of-use and a level of abstraction such that the domain experts or subject matter experts are not required to understand code to design services. The SOA developers can be completely focused on implementing the business logic code behind the service and do not have to worry about SOA infrastructure issues such as versioning and customization.

The Retail Service-Oriented Architecture Enabler Tool fits within the Service Delivery phases. The appropriate use of the tool is after the service analysis phases and the development team is ready for service definition and design. The RSE tool outputs can then be used in the Service Implementation.

RSE is designed to support this type of approach, which is also called top-down Web services development.

Technical Specifications

The Oracle Retail SOA Enabler tool has dependencies on Oracle Retail application installations. This section covers these requirements.

Supported Operating Systems

Supported On	Version Supported
Application Server OS	OS certified with Oracle Fusion Middleware 11g Release1 (11.1.1.6). Options are: ■ Oracle Linux 5 for x86-64 (Actual hardware or Oracle virtual machine). ■ Red Hat Enterprise Linux 5 for x86-64 (actual hardware or Oracle virtual machine) ■ IBM AIX 6.1, 7.1 (actual hardware or LPARs) ■ Solaris 10, 11 Sparc (actual hardware or logical domains) ■ HP-UX 11.31 Integrity (actual hardware, HPVM, or vPars)

Supported On	Version Supported
Application Server	Oracle Fusion Middleware 11g Release 1 (11.1.1.6) **Components:** ■ Oracle WebLogic Server 11g Release 1 (10.3.6) ■ Java: JDK 1.7.0+ 64 bit, or Jrockit 1.6 R28 build or later, within the 1.6 code line. 64 bit. For Linux and Solaris OS only.

2

Installation and Basic Setup

This chapter explains how to deploy the Retail Service-Oriented Architecture Enabler tool to an Oracle WebLogic application server as a Web application.

Installation as a Web Application in Oracle WebLogic

The steps below describe how to deploy the Retail Service-Oriented Architecture Enabler tool to an Oracle WebLogic Application Server as a Web application.

> **Note:** See "Technical Specifications" in Chapter 1.

Prerequisites

The following are prerequisites for installation:

- The retail-soa-enabler-gui.war file is located within the directory structure of the RetailSOAEnabler14.0.0ForAll14.0.0Apps_eng_ga.tar. Locate and extract the contents to a location that is accessible by the browser for deployment.
- The installation and base configuration of the Oracle WebLogic Server is beyond the scope of this document. Work with the Application Server Administration team to determine the physical and logical placement of the retail-soa-enabler-gui component within the WebLogic Server deployment.

> **Note:** See the *Oracle WebLogic Server 11g Release 3 (10.3.6) Installation Guide.*

Deploy the Retail Service-Oriented Architecture Enabler

Using the WebLogic Server Administration Console, complete the following steps:

> **Note:** For instructions with illustrations (screen captures), see "Appendix: Installer Screens."

1. Navigate to the Deployments page.
2. If necessary, click **Lock and Edit** on the left navigation bar to enable the Install button.
3. Click **Install**.

Note: If the application has already been installed, see "Redeploy the Application".

The **Locate deployment to install and prepare for deployment** page is displayed. Follow the instructions to locate the retail-soa-enabler-gui.war file.

4. Select **Upload Files**.
5. On the **Upload a Deployment to the admin server** page, use the Browse button to locate the retail-soa-enabler-gui.war file in the Deployment Archive.
6. Select the retail-soa-enabler-gui.war.
7. Click **Next** and move to **Choose targeting style**.
8. Select **Install this deployment as an application**.
9. Click **Next** and move to Optional Settings.
10. Click **Next** and move to **Select deployment targets**. Select the Server name where you want to install the application.
11. Click **Next** and move to Optional Settings page.
12. In the Security section, select the option **DD only: Use only roles and policies that are defined in the deployment descriptors**.
13. Select **No, I will review the configuration later**.
14. Click **Finish** to deploy the application.
15. Click **Activate Changes** to finish install. Go to Deployments page, select the **retail-soa-enabler-gui** application and click on **Start > Servicing all requests** button. This should change the status of retail-soa-enabler-gui application to **Active** status.
16. After the application is deployed, we need to create a group and users who can access the RSE GUI applications.
17. Go to **Security Realms** page, click on the default realm and go to **Users and Groups** tab.
18. Create a new group named **rseAdminGroup** in the **Groups** page.
19. Go to **Users** page and create a new user.
20. Click on the newly created user and go to the **Groups** tab of that user. From the **Available** groups, select **rseAdminGroup** and move it to the **Chosen** window.
21. Click **Save**.

    This completes the deployment of RSE GUI application

Verify the Retail Service-Oriented Architecture Enabler

1. Navigate to the Deployments page.
2. Locate retail-soa-enabler-gui on the Summary of Deployments page.
3. Click the name, **retail-soa-enabler-gui**, to move to the **Settings for the retail-soa-enabler-gui**.
4. Select the **Testing** tab.
5. Click the **index.jsp URL** in the Test Point.

6. The URL opens the Retail Service-Oriented Architecture Enabler login page.
7. Enter the credentials created in the 'Deploy the Retail Service-Oriented Architecture Enabler' section, and the RSE home page is displayed.
8. The installation is complete. See Chapter 4, "User Interface Usage."

Redeploy the Application

If the retail-soa-enabler-gui application has already been deployed, follow these steps:

1. If the retail-soa-enabler-gui application is running, select **Stop** and **When Work Completes** or **Force Stop Now**, depending on the environment. The recommended option always is **When Work Completes**.
2. Select **Delete**.
3. The retail-soa-enabler-gui should now not show on the Summary of Deployment page.
4. Return to the appropriate step in "Deploy the Retail Service-Oriented Architecture Enabler."

3

Tool Inputs and Outputs

This chapter describes the tool inputs and tool outputs associated with RSE.

Tool Inputs

Tool inputs include the following:

- ServiceProviderDefLibrary.xml
- RestServiceProviderDefLibrary.xml
- XSDs and retail-public-payload-java-beans.jar
- PL/SQL Oracle Objects
- WSDL

ServiceProviderDefLibrary.xml

This is based on ServiceProviderDefLibrary.xsd schema. This definition file contains a high level definition for Provider services for both PL/SQL and Java EE services, and conforms to the ServiceProviderDefinition of a set of services which use Retail Business Objects (BOs) as inputs and outputs.

RestServiceProviderDefLibrary.xml

This is the definition file for Restful Provider services for both PL/SQL and Java EE services, and conforms to the ServiceProviderDefLibrary.xsd schema. This definition file contains a high level definition of a set of services which use Retail Business Objects (BOs) as inputs and outputs.

XSDs and retail-public-payload-java-beans.jar

- The RSE tool references JAXB created java beans based on the BO source schema XSDs. These beans are contained in the retail-public-payload-java-beans.jar.
- The RSE tool will use Oracle Retail BOs from retail-public-payload-java-beans.jar.
- The jar file is located in the WebLogic deployment directory where the RSE tool is deployed.
- The jar file is created using the Retail Artifact Generator from the source BO XSDs.
- The jar file also contains the source XSDs themselves, which will be used by the deployed service to validate all requests and responses against.

PL/SQL Oracle Objects

These artifacts are created from the XSDs using the Retail Artifact Generator. The Objects have to be installed into the database and accessible to the target Web service APIs generated by RSE.

WSDL

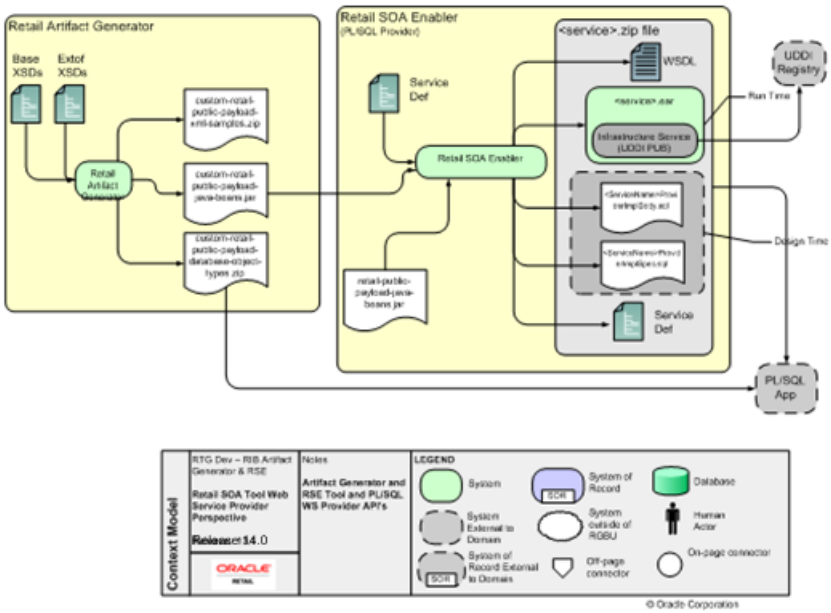
For the Web service consumers, the input is the WSDL of the Web service provider that the service will be consuming.

Tool Outputs

Tool outputs include the following:

- PL/SQL Provider Web service
- PL/SQL Consumer Web service
- Java EE Provider Web service
- Java EE Consumer Web service
- PL/SQL Provider Restful Web service
- Java EE Provider Restful Web service

PL/SQL Provider Web Service



PL/SQL Applications (such as the Oracle Retail Merchandising System) use Oracle Objects, which are similar to the Oracle Retail RIB style APIs. The tool generates a Web service provider layer between the external clients and the PL/SQL APIs to provide the Web service functionality.

The RSE PL/SQL provider output is a zip file. The filename convention is <app>_ PLSQLServiceProvider.zip. For example, rms_PLSQLServiceProvider.zip. The zip file contains the following:

- *<ServiceName>*ProviderImplSpec.sql

This is the specification for the <*ServiceName*>. It creates the package for the <*ServiceName*> in the <*app*> database. It describes all the operations and their IN and OUT parameters for the service.

- <*ServiceName*>ProviderImplBody.sql

  This is the package body for the <*ServiceName*>. This is where the application teams have to write the business logic.

- <*app*>-service.ear

  The.ear file has to be deployed on an Oracle WebLogic. The steps for deployment are in the RSE PL/SQL WS Installation Guide.

- <*app*>-decorator-services.zip

  This zip file contains OSB decorator jars for each service defined in the service definition file. These jars are used by Retail Service Backbone(RSB) as input files.

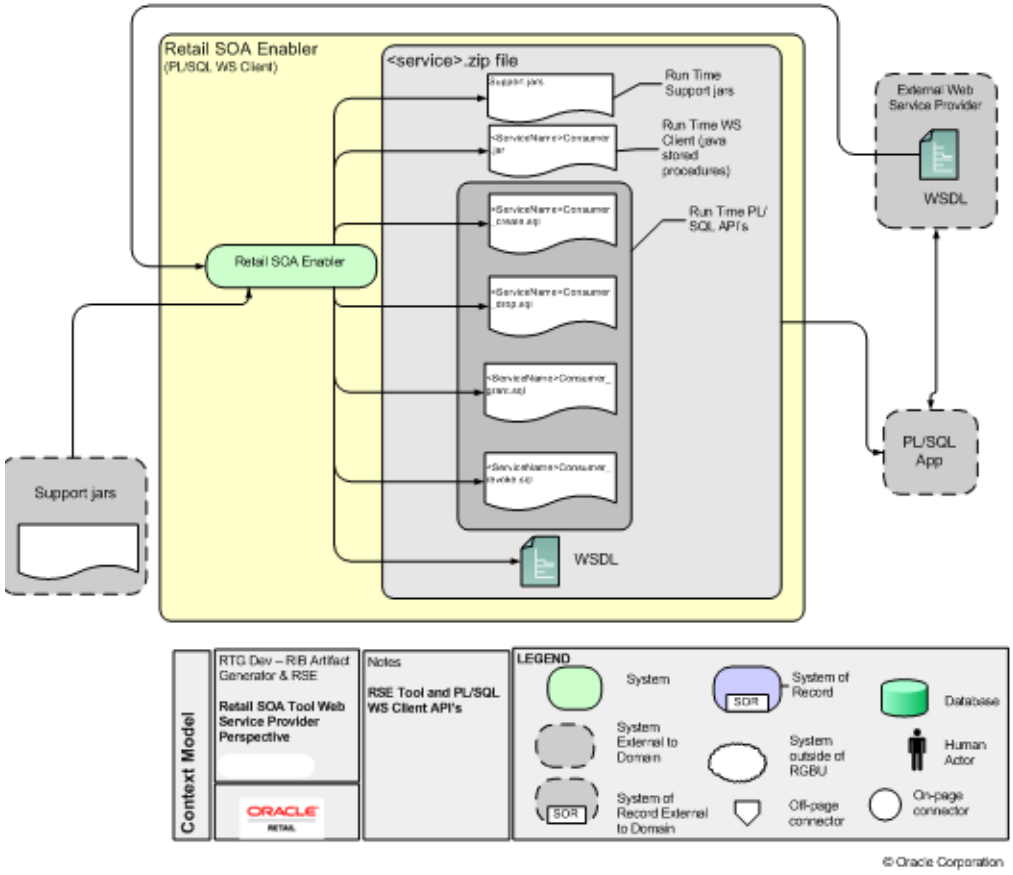
- ServiceProviderDefLibrary.xml

  This is a copy of the ServiceProviderDefLibrary.xml file that was used to create the output.

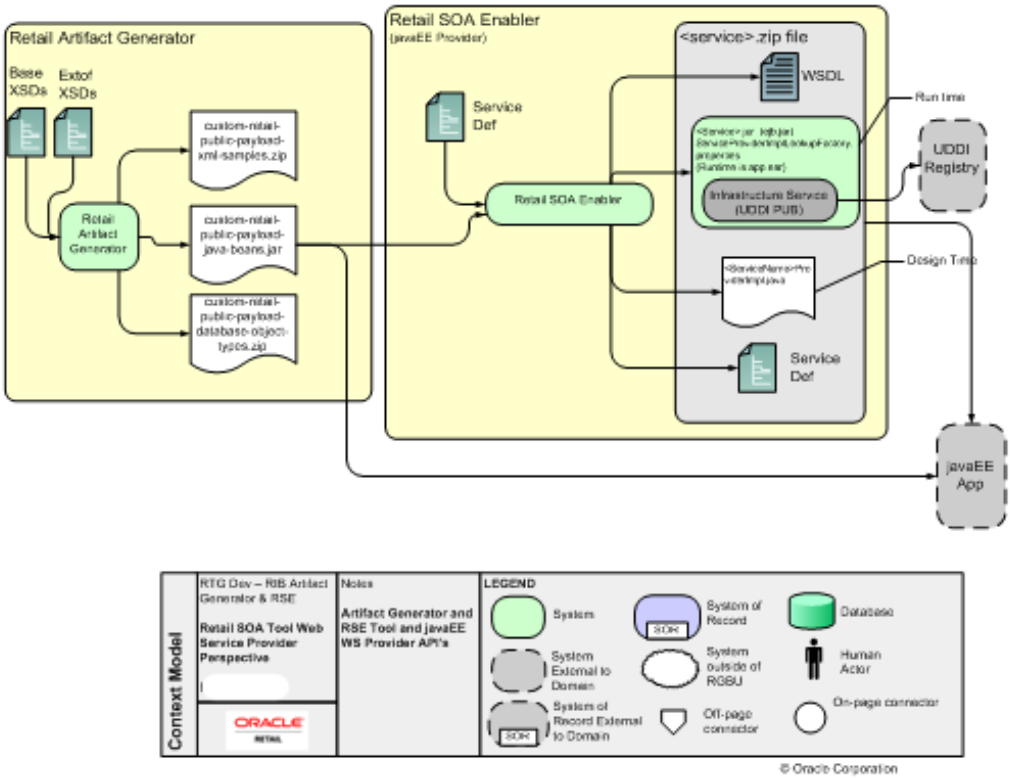
- <*ServiceName*>Service.wsdl

  This is a WSDL file describing the generated Web service. This WSDL file will be fully documented, pulling in documentation elements from both the service def file as well as the BO XSD files. This is a single file with all types inlined. It can be used as input to create a consumer for the generated provider.

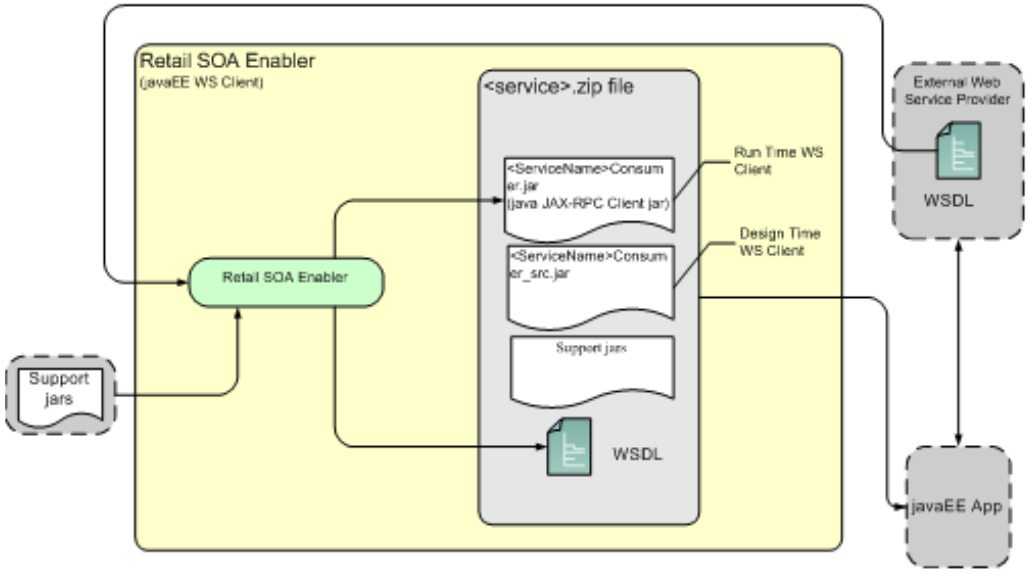
PL/SQL Consumer Web Service

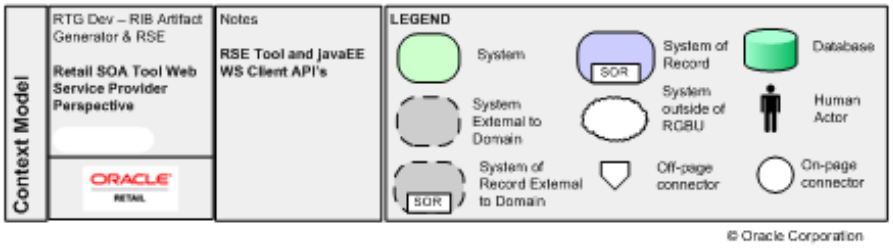


Java EE Provider Web Service



Java EE Consumer Web Service





4

User Interface Usage

The Retail Service-Oriented Architecture Enabler (RSE) tool produces design time and run time artifacts, and it works in conjunction with another tool, the Retail Functional Artifact Generator.

Note: See the *Retail Functional Artifact Generator Guide*.

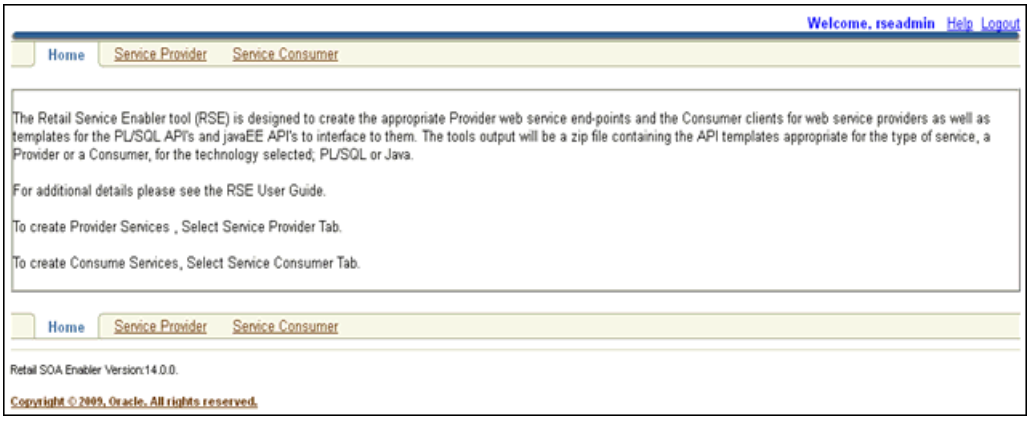
The graphical user interface (GUI) for RSE is hosted on an Oracle WebLogic server as a Web application. Once installed and configured, the GUI is accessed through a URL (http://host:port/contextroot). For example, http://linux1:7001/retail-soa-enabler-gui. First it shows the login page. Here, use the same user name and password to log in which was created in the RSE deployment step and added to the rseAdminGroup. After successful login, it goes to the home page of the application.

The RSE user interface has three tabs, or sections:

- Home
- Server Provider
- Service Consumer

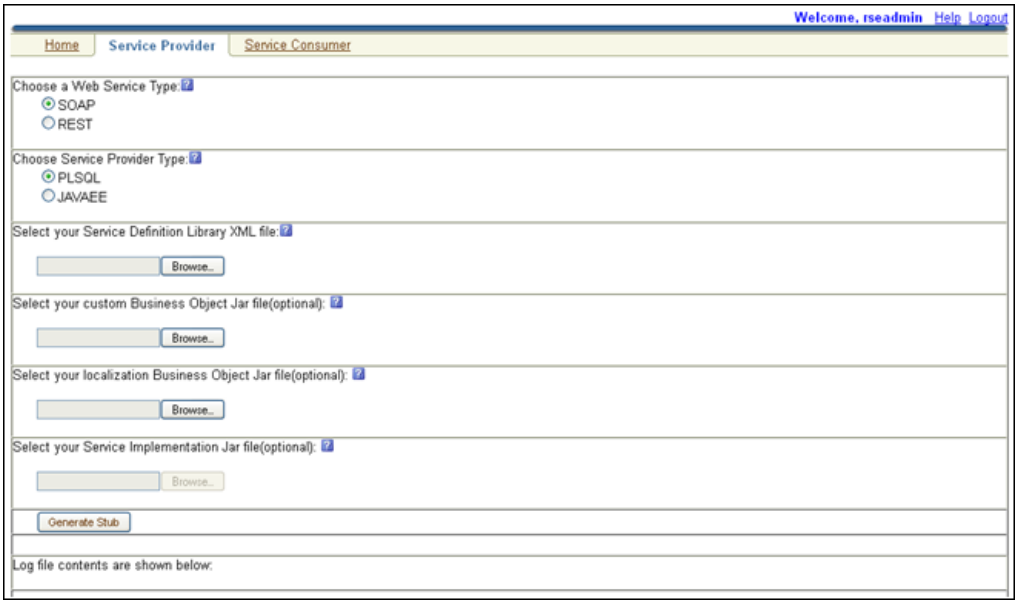
  The user interface is designed to be easy to use. Online help is available, including examples for each function.

  The following is the Home Page.



Service Provider

The service provider screen gives the option of selecting the Web Service Type (SOAP or REST) and Provider type (a Java EE or a PL/SQL service provider).



A PL/SQL service provider can be used by PL/SQL applications such as RMS to expose PL/SQL packages as Web services. The Java EE service provider option allows Java EE applications to create Web services using Oracle Retail payload classes as input and outputs.

The generated Web services do not have any business logic in them. They provide only the framework for the development of Web services.

The inputs for creating Java EE or PL/SQL Web services are as follows:

- Service Definition Library XML file for SOAP web services
- Service Definition Library XML file for RESTful web services
- Custom Business Objects jar file
- Localization Business Object Jar file
- Service Implementation jar file

Service Definition Library XML File

The mandatory input for creating a Java EE or a PL/SQL service provider is a Service Definition Library XML file. This file should contain all the details about the Web services that need to be created.

> **Note:** See Chapter 5," Service Definition Library XML File."

Service Definition Library XML File for Restful web services

The mandatory input for creating a Restful Java EE or a PL/SQL service provider is a Rest Service Definition Library XML file. This file should contain all the details about the web services that need to be created.

Custom Business Objects Jar File

While creating Web services, users may want to use their own payloads to extend the existing payloads. These payloads are known as custom payloads and can be provided to the tool as input for creating Web services. The service provider screen has a field for custom Business Objects jar file. It allows the user to upload a jar file which contains the custom payloads. This jar file is optional; if this is not provided the base payloads are used to create the Web services.

Note: See the *Oracle Retail Functional Artifact Generator Guide* for how to create a custom Business Objects jar file.

Localization Business Object Jar File

While creating Web services, users may want to use localized version of payloads. These payloads are known as localized payloads and can be provided to the tool as an input for creating Web services. The service provider screen has a field for localization Business Object Jar file. It allows the user to upload a jar file which contains the localized payloads. This jar file is optional; if this is not provided, the base payloads are used to create the Web services.

Note: See the *Oracle Retail Functional Artifacts Generator Guide* for how to create a localization Business Objects jar file.

Service Implementation Jar File

This jar file is used only while creating Java EE Web services. While creating Java EE Web services the tool generates empty implementation for the services. Users will have to create their own implementation classes for the Web services and use those classes in the generation of the .ear file in the zip file.

After entering the file names in all the text boxes, click **Generate Stub**.

On successful generation of the stub, the output zip file (<app>_ JavaEEServiceProvider.zip or <app>_PLSQLServiceProvider.zip) will be available as download from the browser. The zip folder contains .ear file which can be deployed on Application Server. In case of RESTful WebService, the ear needs to be deployed on JEE6 compatible server, that is, Glassfish server.

Note: See Chapter 7, "Creating the Java EE Implementation Jar."

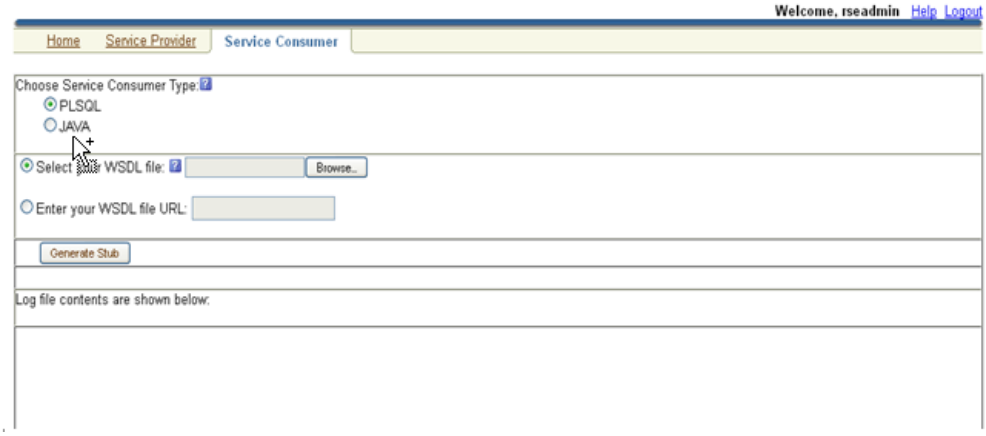
Service Consumer

The Service Consumer tab allows for the creation of a Java EE or PL/SQL service consumer. Service consumer option for restful web services is not supported currently.

Select any one of the option to choose the WSDL file as shown in the screen.

1. Select your WSDL file. Ex: Choose from drive.
2. Enter your WSDL file URL. Example: http/https://<host>:<port>/<ServiceName>Bean/<ServiceName>Service?wsdl
3. Click Generate Stub button to generate the consumer zip file.

When the tool is finished, the consumer distribution zip file can be downloaded to a specific location.



Help

Click the Help link on the right upper corner of the Home page for a brief description of the Service Provider and Service Consumer functionality.

5

Service Definition Library XML File

The Service Definition Library XML file (ServiceDef) is the mandatory input for creating a Java EE or a PL/SQL service provider. This file should contain all the details about the Web services that need to be created.

This chapter provides a detailed description of each section of the schema as well as instructions for managing the Service Definition Library XML file.

Schema Definition

This section discusses the elements of the schema, beginning with the root element and including child elements.

serviceProviderDefLibrary

This is the root element of the schema. The following is an example of the serviceProviderDefLibrary element:

```
<xs:element name="serviceProviderDefLibrary">
       <xs:complexType>
              <xs:sequence>
                     <xs:element ref="service" maxOccurs="unbounded" />
              </xs:sequence>
              <xs:attribute name="appName" type="xs:string" use="required"/>
    <xs:attribute name="version" type="xs:string" use="optional" default="v1"/>
    <xs:attribute name="serviceNamespacePattern" type="xs:string" use="optional"
default="http://www.oracle.com/retail/APPNAME/integration/services/SERVICENAMEServ
ice/VERSION"/>
       </xs:complexType>
</xs:element>
```

Attributes

The serviceProviderDefLibrary has the following attributes:

- **appName**

  This is the name of the application for which the .ear file is being built. When the .ear file is generated, the name of the .ear file starts with the application name. The format of the generated .ear file is *<appName>*-service.ear. For example, if the *appName* is **rms**, the .ear file name is rms-service.ear.

- **serviceNamespacePattern**

  This attribute specifies the pattern for the namespaces that are generated for the Web services. The default value for this attribute is

http://www.oracle.com/retail/APPNAME/integration/services/SERVICENAMEService/VERSION.

- **Version**

  This is the version of the service definition.

Elements

The serviceProviderDefLibrary contains the following elements:

service

Each service element in serviceProviderDefLibrary represents one Web service. The service provider definition should have at least one service defined in it.

The following is an example of the service element:

```
<xs:element name="service">
       <xs:complexType>
              <xs:sequence>
                    <xs:element ref="documentation" minOccurs="0" />
                    <xs:element ref="operation" maxOccurs="unbounded" />
              </xs:sequence>
              <xs:attribute name="name" type="xs:string" use="required"/>
              <xs:attribute name="serviceNamespace" type="xs:string"
use="optional"/>
              <xs:attribute name="serviceVersion" type="xs:string" use="optional"
default="v1"/>
              <xs:attribute name="custom" type="xs:boolean" use="optional"
default="false"/>
      </xs:complexType>
</xs:element>
```

The service element has the following attributes:

- name

  This is the name of the Web service to be created.

- serviceNamespace

  This is the namespace in which the Web service will be created.

- serviceVersion

  This is the version of the Web service. The default value is v1.

The service element contains the following elements:

- Documentation

  This field describes the purpose of the service.

- Operation

  The operation represents the method in the generated Web service. Each service should contain at least one operation.

  The following is an example of the operation element:

  ```
  <xs:element name="operation">
         <xs:complexType>
                <xs:sequence>
                       <xs:element ref="documentation" minOccurs="0" />
                       <xs:element ref="input" />
  ```

```
                        <xs:element ref="output" minOccurs="0" />
                        <xs:element ref="fault" minOccurs="0"
                              maxOccurs="unbounded" />
                 </xs:sequence>
                 <xs:attribute name="name" type="xs:string" use="required"/>
                 <xs:attribute name="suffix" default="inputType">
                <xs:simpleType>
                      <xs:restriction base="xs:string">
                            <xs:enumeration value="inputType" />
                            <xs:enumeration value="outputType" />
                            <xs:enumeration value="NONE" />
                </xs:restriction>
     </xs:simpleType>
</xs:attribute>
<xs:attribute name="custom" type="xs:boolean" use="optional" default="false"/>
</xs:complexType>
</xs:element>
```

The operation element has the following attributes:

- name

  This is the name of the operation.

- suffix

  This is the string to be added to the end of the operation name. One of the following values is supported for this attribute:

  – inputType

    If the suffix value is inputType, the input type name of the operation is added to the generated method name. For example, if the operation name is **create** and input type for that operation name is SupplierDesc, the generated operation name will be createSupplierDesc.

  – outputType

    If the suffix value is outputType, the output type name of the operation is added to the generated method name. For example, if the operation name is **create** and output type for that operation name is SupplierRef, the generated operation name will be createSupplierRef.

  – NONE

    If the suffix value is NONE, a suffix is not added to the operation name.

    Note: If no value is provided for the suffix attribute, inputType is used as the default value.

The operation element contains the following child elements:

- Documentation
- Input
- Output
- Fault

  Fault contains the following elements:

  – Documentation

The description of the fault.

– Faulttype

  The name of the fault.

Managing the Service Definition Library XML File

The Service Definition Library XML file is the single source of truth for the RSE tool. This section discusses the creation and management of the file.

Creating the File

The Service Definition Library XML example in "Appendix: Sample ServiceProviderDefLibrary.xml" can be used as the initial template. Use the instructions in the Service Definition Library XML File section to construct the ServiceDef according to the goals of the service requirements.

As discussed in the Concepts section, the creation of this file is the result of the analysis phase and part of the Service Design phase. The template provides the placeholders for the standard service components: service name, operation name, and the contracts for each of the operations, as well as the standard faults.

The ServiceDef should be created and managed (or governed) as a service-oriented architecture asset in a source code control system. It is as important as the Service Contracts (XSDS) and implementation source code.

Changing the Version of the File

To change the version of the service definition library file, a **version** attribute must be added to the root element, serviceProviderDefLibrary.

For example:

```
<serviceProviderDefLibrary appName="rms"
xmlns=http://www.oracle.com/retail/integration/services/serviceProviderDefLibrary/
v1
version="v2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...

</serviceProviderDefLibrary>
```

Changing the appName Attribute in the File

To change the application name in the services, edit the *appName* attribute in the root element, serviceProviderDefLibrary.

For example:

```
<serviceProviderDefLibrary appName="editThisAppName"
xmlns=http://www.oracle.com/retail/integration/services/serviceProviderDefLibrary/
v1 version="v2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
...

</serviceProviderDefLibrary>
```

Renaming a Service or Operation Name in the File

To rename a service, edit the name attribute in the service element.

For example:

```
<serviceProviderDefLibrary appName="rms"
xmlns=http://www.oracle.com/retail/integration/services/serviceProviderDefLibrary/
v1
version="v2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<service name="EditThisName">
...
</serviceProviderDefLibrary>
```

To rename an operation in the service, edit the name attribute of the operation element.

Adding a New Service or New Operation to the File

To add a new service to library, add a new service element with its child elements.

For example:

```
<serviceProviderDefLibrary appName="rmscostchange"
xmlns=http://www.oracle.com/retail/integration/services/serviceProviderDefLibrary/
v1
version="v2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

      <service name="ExistingService">
              <operation name="existingOperation">
                     <documentation></documentation>
                     <input type="XXX">
                             <documentation></documentation>
                     </input>
                     <output type="YYY">
                            <documentation></documentation>
                     </output>
                     <fault faultType="IllegalArgumentWSFaultException">
                             <documentation>Throw this exception when a
"soap:Client" side message problem occurs.</documentation>
                      </fault>
                      <fault faultType="EntityAlreadyExistsWSFaultException">
                             <documentation>Throw this exception when the attempt
made to  create a object that already exists.</documentation>
                       </fault>
                       <fault faultType="IllegalStateWSFaultException">
                               <documentation>Throw this exception when an unknown
            "soap:Server" side problem ccurs.</documentation>
                       </fault>
             </operation>
</service>
<service name="AddedNewServiceName">
             <operation name="Operation">
                    <documentation></documentation>
                    <input type="XXX">
                           <documentation></documentation>
                    </input>
                    <output type="YYY">
                           <documentation></documentation>
```

```
                    </output>
                    <fault faultType="IllegalArgumentWSFaultException">
                            <documentation>Throw this exception when a
soap:Client" side message problem occurs.</documentation>
                     </fault>
                     <fault faultType="EntityAlreadyExistsWSFaultException">
                             <documentation>Throw this exception when the attempt
made to create a object that already exists.</documentation>
                     </fault>
                     <fault faultType="IllegalStateWSFaultException">
                             <documentation>Throw this exception when an unknown
                                   "soap:Server" side problem
occurs.</documentation>
                    </fault>
              </operation>
</service>


...

</serviceProviderDefLibrary>
```

To add a new operation to a service, add the operation element with its child elements.

For example:

```
<service name="service">
              <service name="ServiceName">
              <operation name="NewAddedOperation">
                     <documentation></documentation>
                     <input type="XXX">
                            <documentation></documentation>
                     </input>
                     <output type="YYY">
                            <documentation></documentation>
                     </output>
                     <fault faultType="IllegalArgumentWSFaultException">
                             <documentation>Throw this exception when a
"soap:Client" side message problem occurs.</documentation>
                     </fault>
                     <fault faultType="EntityAlreadyExistsWSFaultException">
                            <documentation>Throw this exception when the attempt
made to create a object that already exists.</documentation>
                     </fault>
                     <fault faultType="IllegalStateWSFaultException">
                             <documentation>Throw this exception when an unknown
"soap:Server" side problem ccurs.</documentation>
                     </fault>
              </operation>
              <operation name="ExistingOperation">
                    <documentation></documentation>
                    <input type="XXX">
                           <documentation></documentation>
                    </input>
                    <output type="YYY">
                           <documentation></documentation>
                    </output>
                    <fault faultType="IllegalArgumentWSFaultException">
                             <documentation>Throw this exception when a
soap:Client" side message problem occurs.</documentation>
                     </fault>
```

```
                    <fault faultType="EntityAlreadyExistsWSFaultException">
                            <documentation>Throw this exception when the attempt
made to create a object that already exists.</documentation>
                        </fault>
                        <fault faultType="IllegalStateWSFaultException">
                                <documentation>Throw this exception when an unknown
                                      "soap:Server" side problem
occurs.</documentation>
                        </fault>
                </operation>
</service>
```

Deleting a Service or Deleting Operations from the File

To delete a service from the library, remove the service element and all its child elements from the library.

To delete an operation from the service, delete the operation element and all its child elements.

Defining New Exceptions to the Operations

Users can define a new exception in the service definition library. The RSE tool creates the artifacts with this new exception.

For example:

```
<operation name="ExistingOperation">
                    <documentation></documentation>
                    <input type="XXX">
                           <documentation></documentation>
                    </input>
                    <output type="YYY">
                           <documentation></documentation>
                    </output>
                    <fault faultType="IllegalArgumentWSFaultException">
                            <documentation>Throw this exception when a
"soap:Client" side  message problem occurs.</documentation>
                    </fault>
                    <fault faultType="EntityAlreadyExistsWSFaultException">
                            <documentation>Throw this exception when the attempt
made to create a object that already exists.</documentation>
                    </fault>
                    <fault faultType="IllegalStateWSFaultException">
                            <documentation>Throw this exception when an unknown
"soap:Server" side problem  occurs.</documentation>
                    </fault>
                    <fault faultType="UserDefinedException">
                            <documentation>This is user defined exception for a
particular scenerio.</documentation>
                    </fault>
             </operation>
```

Using Different Versions of Objects as Input/Output to an Operation

The version difference between objects does not impact the RSE tool, as long as the objects adhere to standards.

6

Service Definition Library XML File for Restful services

The REST Service Definition Library XML file (ServiceDef) is the mandatory input for creating a RESTful Java EE or a PL/SQL service provider. This file should contain all the details about the Web services that need to be created.

This chapter provides a detailed description of each section of the schema as well as instructions for managing the Service Definition Library XML file.

Schema Definition

This section discusses the elements of the schema, beginning with the root element and including child elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.oracle.com/retail/integration/services/serviceProvider
DefLibrary/v1"

xmlns="http://www.oracle.com/retail/integration/services/serviceProviderDefLibrary
/v1"
    elementFormDefault="qualified">
    <xs:element name="serviceProviderDefLibrary">

<xs:complexType>
<xs:sequence>
                <xs:element ref="service" maxOccurs="unbounded" />
            </xs:sequence>
            <xs:attribute name="appName" type="xs:string" use="required"/>
            <xs:attribute name="version" type="xs:string" use="optional"
default="v1"/>
            <xs:attribute name="serviceNamespacePattern" type="xs:string"
use="optional"
default="http://www.oracle.com/retail/APPNAME/integration/services/SERVICENAMEServ
ice/VERSION"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="documentation" type="xs:string" />

<xs:element name="header">
<xs:complexType>
            <xs:attribute name="type" type="xs:string" use="required"/>
            <xs:attribute name="headerName" type="xs:string" use="required"/>
            <xs:attribute name="headerRequired" type="xs:boolean" use="optional"
```

```
default="false"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="input">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="documentation" minOccurs="0" />
                <xs:element ref="header" minOccurs="0" maxOccurs="unbounded"  />
            </xs:sequence>
            <xs:attribute name="type" type="xs:string" use="required"/>
            <xs:attribute name="version" type="xs:string" use="optional"
default="v1"/>
            <xs:attribute name="custom" type="xs:boolean" use="optional"
default="false"/>
            <xs:attribute name="identifierNameList" type="xs:string"
use="optional" />
        </xs:complexType>
    </xs:element>
    <xs:element name="output">

<xs:complexType>
<xs:sequence>
                <xs:element ref="documentation" minOccurs="0" />
                <xs:element ref="relations" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="type" type="xs:string" use="required"/>
            <xs:attribute name="version" type="xs:string" use="optional"
default="v1"/>
            <xs:attribute name="custom" type="xs:boolean" use="optional"
default="false"/>

        </xs:complexType>
    </xs:element>
    <xs:element name="relations">

<xs:complexType>
<xs:sequence>
        <xs:element ref="relatedTo" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>

    <xs:element name="relatedTo">
    <xs:complexType>
        <xs:attribute name="name" use="required">
            <xs:simpleType>
                    <xs:restriction base="xs:string">
     <xs:pattern value="[a-zA-Z0-9]*|EXTERNAL_SYSTEM"></xs:pattern>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
        <xs:attribute name="value" type="xs:string" use="optional" />
            </xs:complexType>
            </xs:element>



<xs:element name="fault">
<xs:complexType>
            <xs:sequence>
```

```
                <xs:element ref="documentation" minOccurs="0" />
            </xs:sequence>
            <xs:attribute name="faultType" type="xs:string" use="required"/>
        </xs:complexType>
    </xs:element>
    <xs:element name="operation">

<xs:complexType>
<xs:sequence>
                <xs:element ref="documentation" minOccurs="0" />
                <xs:element ref="input" />
                <xs:element ref="output" minOccurs="0" />
                <xs:element ref="fault" minOccurs="0"
                    maxOccurs="unbounded" />
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="suffix" default="inputType">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="inputType" />
                        <xs:enumeration value="outputType" />
                        <xs:enumeration value="NONE" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="custom" type="xs:boolean" use="optional"
default="false"/>
            <xs:attribute name="operationType" default="READ_WITH_IDENTITY">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="CREATE" />
                        <xs:enumeration value="READ_WITH_IDENTITY" />
                        <xs:enumeration value="READ_WITH_PREDICATE" />
                        <xs:enumeration value="UPDATE" />
                        <xs:enumeration value="DELETE" />
                        <xs:enumeration value="PROCESS" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
        </xs:complexType>
    </xs:element>
    <xs:element name="service">

<xs:complexType>
<xs:sequence>
                <xs:element ref="documentation" minOccurs="0" />
                <xs:element ref="operation" maxOccurs="unbounded" />
            </xs:sequence>
            <xs:attribute name="name" type="xs:string" use="required"/>
            <xs:attribute name="serviceNamespace" type="xs:string"
use="optional"/>
            <xs:attribute name="serviceVersion" type="xs:string" use="optional"
default="v1"/>
            <xs:attribute name="custom" type="xs:boolean" use="optional"
default="false"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

ServiceProviderDefLibrary

This is the root element of the schema. The following is an example of the serviceProviderDefLibrary element.

Sample Rest Service Definition Library file:

```
<serviceProviderDefLibrary appName="rms"
xmlns="http://www.oracle.com/retail/integration/services/serviceProviderDefLibrary
/v1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<service name="Suppliers">
      <operation name="find" operationType="READ_WITH_PREDICATE"
suffix="outputType">
         <input type="SupplierColRef" identifierNameList="country_id"/>
         <output type="SupplierColDesc">
         </output>
         <fault faultType="IllegalArgumentWSFaultException" />
      </operation>
      <operation name="delete" operationType="DELETE">
         <input type="SupplierColRef" identifierNameList="country_id"/>
         <output type="SupplierColRef">
         </output>
         <fault faultType="IllegalArgumentWSFaultException" />
      </operation>

</service>
</serviceProviderDefLibrary>
```

Validation rules for a service definition xml for RESTful web services

- There must be a valid appName specified in service def. It must follow these rules:
  - must not be empty.
  - must be alphanumeric.
  - must not start with a number.
- There must be at least one service in the service def.
- The service name must be unique in the service def.
- The service name must follow these rules:
  - must not be empty.
  - must be alphanumeric.
  - must not start with a number.
- There must be at least one operation in the service.
- For each operation a valid OperationType must be defined. Operation type can be one of the following:
  - CREATE
  - UPDATE
  - DELETE
  - PROCESS
  - READ_WITH_IDENTITY

- READ_WITH_PREDICATE

- A service must have an operation of either READ_WITH_IDENTITY or READ_WITH_PREDICATE type. If the operation type is READ_WITH_IDENTITY, then the service will generate methods to work with one object. If it operation is READ_WITH_PREDICATE, then the web service is for collection of objects.
- The operation name must follow these rules:
  - must not be empty.
  - must be alphanumeric.
  - must not start with a number.
- Each operation must have a valid input type and output type defined. These types must be the names of valid payload objects.
- There can be multiple methods of operation type PROCESS for a service.
- A service can have either READ_WITH_IDENTITY operation or READ_WITH_PREDICATE, it cannot have both.
- READ_WITH_PREDICATE operation type is only supported for CREATE and PROCESS operations. DELETE and UPDATE operations are not supported on a collection of objects.
- The input type of an operation must have a valid value for the field identifierNameList, which means that the value specified for that field must follow these rules:
  - It can have comma-separated names of fields.
  - The fields must be valid elements present in the xsd of the payload name specified in inputType of that operation.
  - If the service has an operation of type READ_WITH_PREDICATE, then the payload specified in inputType must have an element named "collection_size".That signifies that this service is for collection of objects.
- The identifierNameList specified in READ_WITH_IDENTITY is used for building URI for the service. For example, if the service name is "Supplier" and identifierNameList for READ_WITH_IDENTITY operation has a value such as "supplier_id, sup_xref_key" then the URI of that service will be http://<host>:<port>/<contextPath>/SupplierResource/<supplierId>/<sup_xref_key>. An example of a request is: http://localhost:7001/rms-service/1/xref1.In this example, "1" will be substituted as the value for supplierId and "xref1" will be substituted as the value for "sup_xref_key" field.
- All the operations of a service should contain the same value for identifierNameList. The identifierNameList specified in READ_WITH_IDENTITY operation is used for building the URI for UPDATE, DELETE and PROCESS methods also for the service, because the URI represents the object it is working on, and the READ_WITH_IDENTITY, DELETE, UPDATE and PROCESS methods should work on the same object.
- A service can have relations sepcified in the outputType of the operation. The name specified in relatedTo element must be the name of a service which exists in the same service def xml.
- A service cannot have its own name in relatedTo field

7

Web Service Standards and Conventions

This chapter includes standards and conventions for Web service naming and versioning.

Web Service Naming

The following standards and conventions apply to the naming of Web services.

The Web service name should be a business noun, concept or process.

Item	Description
Recommendation	The Web service name should be a business noun, a business concept, or a business process.
Rationale	To be in alignment with other Web service standards.
Example	Supplier Service

Avoid verbs when naming Web services.

Item	Description
Recommendation	The Web service name should be a business noun, a business concept, or a business process.
Rationale	Verbs generally are at the operation level, not at the service level.
Example	Avoid names such as CreateSupplierService.

The first 30 characters of the Web service name must be unique.

Item	Description
Recommendation	The first 30 characters of the Web service name must be unique.
Rationale	Some systems truncate names at 30 characters.
Example	N/A

The integration/services qualifier should be in the namespace.

Item	Description
Recommendation	The integration/services qualifier should be in the namespace.

Item	Description
Rationale	
Example	http://www.oracle.com/retail/rms/integration/services/PayTermService.

The Web service namespace should contain the application short name.

Item	Description
Recommendation	The Web service namespace should contain the application short name.
Rationale	Multiple applications may publish services with similar names. To categorize and identify which application is hosting what service, the service namespace should contain the application short name.
Example	http://www.oracle.com/retail/rms/integration/services/PayTermService.

The Web service type should be document/literal wrapped.

Item	Description
Recommendation	The Web service type should be **document/literal wrapped**.
Rationale	This is defined in the WSDL.
Example	`<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/> <operation name="createPayTermBO"> <ns21:PolicyReference xmlns:ns21="http://www.w3.org/ns/ws-policy" URI="#PayTermServicePortBinding_createPayTermBO_WSAT_ Policy"/> <soap:operation soapAction=""/> <input> <soap:body use="literal"/> </input>`

The Web service must comply with Web Service Basic Profile 1.1.

Item	Description
Recommendation	The Web service must comply with Web Service Basic Profile 1.1.
Rationale	The specification is called the WS-I Basic Profile 1.1. It consists of a set of non-proprietary Web services specifications, clarifications, refinements, interpretations, and amplifications of those specifications which promote interoperability.
Example	N/A

The Web service operation naming pattern should be verb<TopLevelComplexType>(TopLevelComplexType variable).

Item	Description
Recommendation	The operation name pattern should be either of the following: ■ verb<TopLevelComplexType>(TopLevelComplexType variable) ■ verb<NonTopLevelComplexType>Using<TopLevelComplexType>(TopLevelComplexType variable).
Rationale	The operation name should reflect the Top Level Complex Type of the service's primary entity object to ensure the name is unambiguous.
Example	createItemListBO

Web Service Versioning

Service versioning is in the namespace, including the application and the version identifier.

The service namespace is versioned.

Item	Description
Recommendation	The WSDL for the RBS will have the namespace versioned.
Rationale	For breaking changes only, the WSDL for the RBS will have the namespace versioned. http://www.oracle.com/retail/<retail app>/integration/services/<service name>/V<incremental change number>
Example	http://www.oracle.com/retail/rms/integration/services/PayTermService/V2

8

Creating the Java EE Implementation Jar

Creating Web services with different implementations is a three-step process, as described below.

Note: For creating an implementation class, interface classes are required.

Step 1: Generate Web Services with Default Implementation

Generate Web services with the default implementation as follows:

1. Provide the Service Definition Library XML file and click **Generate Stub** to create a zip file.
2. The zip file contains a jar file with the interface classes for the Web services. The name pattern of the jar file is <*appName*>-service-ejb.jar.

   For example, if the application name in ServiceDef is rms, the jar file name is rms-service-ejb.jar.

   The jar file also contains a properties file named ServiceProviderImplLookupFactory.properties. This file contains the name of the Web service interface and the class implementing the Web service.

Step 2: Implement Interfaces

Implement the interfaces and create the implementation classes. The classes can be packaged in a jar file. Upload the jar file while creating the final ear file.

Step 3: Upload the jar

When using the Service Implementation Jar File option to upload the jar containing the implementations, the default service implementation jar is not included in the .ear file. Rather, the jar file provided by the user is included. When the Web service is invoked, the service implementation provided by the user is invoked.

9

Implementation Guidelines

This chapter provides a set of implementation notes that may be helpful when implementing the Oracle Retail Service-Oriented Architecture Enabler (RSE) tool. The information included here is intended to provide guidance on the following topics:

- PL/SQL Service Consumer
- PL/SQL Provider Service
- Java EE Web Service Consumer
- Java EE Service Provider
- Web Service Call as a Remote EJB Call
- Web Service Call as a POJO Call
- Deploying the Web Service
- Creating a JDBC Data Source

Important Note About this Chapter

The implementation notes in this chapter are intended to provide some guidance in the development and deployment of the Web service layer. This information does not take into account the implementation of the business logic required to complete the application API layer.

The RSE tool and approaches described in this section are complex. A high level of skill and knowledge of the product is required to complete these implementation tasks. Also required is technology specific development of application APIs and the business logic that is needed to complete it.

Any issues that may arise with development tools, development environments, custom APIs, or custom message flows are the responsibility of the customer and not Oracle Retail.

PL/SQL Service Consumer Implementation Notes

To set up the Web service consumer side proxies, complete the following steps:

> **Note:** See the section, "Important Note About this Chapter".

1. loadjava -u *<username>*/*<password>*@*<host>*:*<port>*:*<SID>* -r -v -f -genmissing dbwsclientws.jar dbwsclientdb102.jar

Note: **loadjava i**s a utility available in Oracle Database.

2. Edit and run *_grant.sql script as sysdba to give the user proper permission.
3. Load the following jars to the database.

   Instructions to load jars to database can be found in PLSQLServiceConsumer_ ReadMe.doc packaged with the generated zip file.

   - xmlparserv2.jar
   - dbwsa.jar
   - dbwsclientdb11.jar
   - dbwsclientws.jar
   - <WebServiceName>ServiceConsumer.jar
   - http_client.jaradMe.doc packaged with the generated zip file

   Note: The ojdbc6.jar should not be loaded, because it is used only for loading the other jars. If the jar is already loaded, drop the jar. If you get ORA-29533 while dropping the jar, drop the individual files.

   For example:

   ```
   dropjava -u
   <username>/<password>@<host>:<port>:<SID>packageName/SourceName
   ```

4. Run the *Consumer_create.sql in the schema that will use this API. The schema owner is the user granted permission in Step 2.
5. Write a PL/SQL procedure to work as the client to call the Web service. A sample is provided below:

   Note: The following sample code is written for the PayTerm Web service. Replace the service endpoint URL and the consumer class name according to the Web service for which the client is generated.

```
create or replace PROCEDURE wstestClient IS
BEGIN
PayTermServiceConsumer.setEndpoint('http://10.141.26.93:7001/PayTermBean/PayTer
mService');
dbms_output.PUT_LINE(PayTermServiceConsumer.getEndPoint());
dbms_output.PUT_LINE(PayTermServiceConsumer.ping('TestMessage'));
dbms_output.PUT_LINE('Done.');
END;
```

PL/SQL Provider Service Implementation Notes

The distribution (.zip) file includes an .ear file that contains all the generated code for the service; it is ready to be deployed to the application server. The business logic can be implemented in PL/SQL packages in Oracle. The distribution contains the "spec" and body scripts for the packages called by the deployed service.

To complete implementation, follow these steps:

Note: See the section, "Important Note About this Chapter".

1. Create the PL/SQL service provider distribution file using the RSE tool. The output of this process is the .zip file.

   Note: See Chapter 4,"User Interface Usage".

2. Extract the <service_name>.ProviderImplSpec.sql and <service_name>ProviderImplBody.sql files from the distribution zip file.
3. These files will be modified to provide a PL/SQL implementation for the service.
4. Extract the <service_name>-service.ear file from the distribution zip file. This file is the generated Web service that will be deployed.
5. Create the JDBC data source.

   Note: See "Creating a JDBC Data Source".

6. If not already deployed, deploy the Oracle Objects to the appropriate database user.

   Note: See the *Oracle Retail Functional Artifact Generator Guide*.

7. Modify the PL/SQL body file for the business logic implementation. The <service_name>ProviderImplBody.sql file contains comments about where to implement logic for each method on the service.
8. Install the modified PL/SQL packages to the database. They will be called by the Web service methods.
9. Deploy the <service_name>.ear file to the Oracle WebLogic Server.

Java EE Service Consumer Implementation Notes

The Java Web service consumer artifacts generated by this tool are based on the JAX-WS 2.1 specification. Services can be invoked in synchronous and asynchronous mode by using these artifacts.

To complete implementation, follow these steps:

Note: See the section, "Important Note About this Chapter".

1. Create a Web service client.
2. Create the application that uses the {WebsSrviceName}ServiceConsumer.jar and code your Web service client. The {WebsSrviceName}ServiceConsumer.jar contains all necessary code to invoke the {WebsSrviceName}Service WebService.
3. Additional JAX-WS library jars might be required.
4. Deploy the service in the server.

5. Invoke the Web service client to see the results.

Sample Client Code

The code below is an example of how to invoke Oracle Retail's PayTerm Web service. For each Web service, a specific WebServiceConsumer code/jar must be generated that can "talk to" the service.

> **Note:** The following sample code is for invoking the PayTerm Web service. When you generate Java consumer for a Web service, the generated jar file will contain classes specific to that Web service. Use the appropriate classes in the client code. Service namespace and WSDL location also should be changed accordingly.

```
import java.math.BigDecimal;
import java.net.URL;
import javax.xml.namespace.QName;
import com.oracle.retail.integration.base.bo.paytermdesc.v1.PayTermDesc;
import com.oracle.retail.integration.base.bo.paytermref.v1.PayTermRef;
import
com.oracle.retail.rms.integration.services.paytermservice.v1.PayTermPortType;
import
com.oracle.retail.rms.integration.services.paytermservice.v1.PayTermService;
import junit.framework.TestCase;

public class PayTermTest extends TestCase{
   public void testCreatePayTerm(){
          try{
                  //qname is the namespace of the web service
                  QName qName = new
QName("http://www.oracle.com/retail/rms/integration/services/PayTermService/v1",
"PayTermService");

                  //wsdlLocation is the URL of the WSDL of the web service
                  URL wsdlLocation = new
URL("http://10.141.26.93:7001/PayTermBean/PayTermService?WSDL");

                  //get the web service instance
                  PayTermService service = new PayTermService(wsdlLocation,qName);
                  PayTermPortType port = service.getPayTermPort();

                  //populate input object for the web service method
                  PayTermDesc desc = new PayTermDesc();
                  desc.setTerms("terms");
                  desc.setDiscdays("1");
                  desc.setDueDays("1");
                  desc.setEnabledFlag("t");
                  desc.setPercent(new BigDecimal("1"));
                  desc.setRank("1");
                  desc.setTermsCode("code");
                  desc.setTermsDesc("desc");
                  desc.setTermsXrefKey("key");

                  //call the web service method. here ref is the response object
of the web service.
                  PayTermRef ref =  port.createPayTermDesc(desc);

           }catch(Exception e){
```

```
                    e.printStackTrace();
                }
            }
        }
```

Java EE Service Provider Implementation Notes

The RSE tool creates the appropriate provider Web service end-points as well as a skeleton implementation layer where the developer implements business logic. All of this is packaged inside the provider distribution archive file.

The Java EE Provider distribution file provides a sample deployable application and all the libraries that can be used to create Web services using retail payloads. The distribution file follows the naming convention of <*appname*>_ JavaEEServiceProvider.zip. For example, the distribution file for the RMS application is named rms_JavaEEServiceProvider.zip. The <rms> prefix must be replaced with the name of any other application being developed.

The Web services generated by the RSE tool can be implemented and deployed in a number of ways. This section includes three implementation use cases for reference.

Note: See the section, "Important Note About this Chapter".

Use Case 1: Complete the Generator Provided Stub Code Implementation

1. Generate the distribution file using the RSE tool.
2. Extract the <service_name>-ejb-impl-src.jar file from the zip file.
3. Extract the <service_name>-service.ear file from the zip file.
4. Add business logic code where indicated in the Impl java files.
5. Use the java jar command to re-build the <service_name>-service-ejb-impl.jar file.
6. Use the jar command to update .ear file with the new implementation jar.
7. Deploy the .ear file to the server.

Use Case 2: Provide a Custom impl jar to the RSE Tool

1. Create custom java classes that implement the <service_name>ServiceProvider interfaces contained in the <service_name>-service-ejb.jar file.
2. Extract the ServiceProviderImplLookupFactory.properties file from the .ear file.
3. Modify the properties file to point to your implementation classes for the services.
4. Use the jar command to create a jar containing your implementation classes, as well as the modified properties file.
5. Run the RSE tool again and provide the new custom implementation jar file.
6. Extract and deploy the generated .ear file to the server.

Use Case 3: Package the Generated Service Classes in an Existing Application

1. Generate the distribution file using the RSE tool.
2. The service interfaces are provided in the <*appname*>-service-ejb.jar file in the distribution file. This jar file should be included in the application classpath.

3. Source code of sample implementations for the service interfaces are provided in the <*appname*>-service-ejb-src.jar file in the distribution file. (If application developers want to use the same classes in their application, they can extract the java files from the jar file and include those in application source code. They also can add their own business logic in the method implementations. If they decide to write their own implementations, they should make sure that the appropriate service interfaces are implemented.)
4. After writing the Web service implementations, the java files should be compiled. The class files can be included in a new jar file or in the same jar file used for the rest of the classes of the application.
5. Modify the ServiceProviderImplLookupFactory.properties file to include appropriate class names of service implementations and include it in application classpath. A recommended approach is to include the properties file in the jar file that contains the service implementation classes.
6. Make sure that the following jar files are included in the application ear file:
   - <*appname*>-service-ejb.jar
   - Jar file containing the service implementation classes
   - jaxb-api.jar
   - retail-public-payload-java-beans-base.jar
   - retail-public-payload-java-beans.jar
   - retail-soa-enabler.jar
7. Include an ejb-module in the application.xml of the application. The module name should be same as the name of <*appname*>-service-ejb.jar file.
8. The .ear file is ready for deployment on the server.

Web Service Call as a Remote EJB Call

This section applies to PL/SQL Web service implementations and Java EE Web service implementations.

A client can call a Web service as a remote EJB call to improve performance by avoiding marshalling and unmarshalling.

Note: See the section, "Important Note About this Chapter".

Prerequisites

The following is a list of prerequisites to implementation.

1. Get the updated wlfullclient.jar (integration-lib\third-party\oracle\wl\10.3\)& retail-soa-enabler.jar (integration-lib\internal-build\rse\) from the Repository.
2. Run build.xml for retail-soa-enabler.
3. Generate the .ear and deploy it to server.
4. Configure the data source in the server.

Procedure

Complete the following steps.

1. Create a Java file containing the code below inside any package. (See code sample at the end of this section.)
2. Include the following jar files in the classpath:
   - retail-public-payload-java-beans-base.jar
   - retail-public-payload-java-beans.jar
   - oo-jaxb-bo-converter.jar
   - retail-soa-enabler.jar
   - <*appname*>-service-ejb.jar
3. Run code as a Java application.

> **Note:** The sample code below obtains a context for accessing the WebLogic naming service and calls a lookup method to get the Object inside the container by providing a binding name. It then calls a corresponding Web service method. As an example, the code sample calls the PayTerm service.

```
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.oracle.retail.integration.base.bo.paytermdesc.v1.PayTermDesc;
import com.oracle.retail.integration.base.bo.paytermref.v1.PayTermRef;
import
com.oracle.retail.integration.services.exception.v1.EntityNotFoundWSFaultException
;
import
com.oracle.retail.integration.services.exception.v1.IllegalArgumentWSFaultExceptio
n;
import
com.oracle.retail.integration.services.exception.v1.IllegalStateWSFaultException;
import com.oracle.retail.rms.integration.services.paytermservice.v1.PayTermRemote;


public class WebLogicEjbClient {

        public static void main(String[] args) throws NamingException,
IllegalArgumentWSFaultException, EntityNotFoundWSFaultException,
IllegalStateWSFaultException {

                Context ctx = getInitialContext("t3://localhost:7001", "<WLS
user>","<WLS password>");
Object ref = ctx .lookup("PayTerm#com.oracle.retail.rms.integration.services.
paytermservice.v1.PayTermRemote");

                PayTermRemote remote = (PayTermRemote)(ref);

                PayTermRef ref = new PayTermRef();
                PayTermDesc desc = remote.findPayTermDesc(ref);

                System.out.println("findPayTermDesc=" + desc);

}

static Context getInitialContext(String url, String user, String password)
```

```
throws NamingException {

      Properties h = new Properties();
      h.put(Context.INITIAL_CONTEXT_FACTORY,
      "weblogic.jndi.WLInitialContextFactory");
      h.put(Context.PROVIDER_URL, url);
      h.put(Context.SECURITY_PRINCIPAL, user);
      h.put(Context.SECURITY_CREDENTIALS, password);
      return new InitialContext(h);

     }
}
```

Code Description

Code sample 1:

```
Context ctx = getInitialContext("t3://localhost:7001", "<WLS user>","<WLS
password>");
```

Description: Gets Initial Context object by passing the URL (local WebLogic URL, if not configured to other), user name, and password of the server.

Code sample 2:

```
Object ref = ctx .lookup("PayTerm#com.oracle.retail.rms.integration.services.
paytermservice.v1.PayTermRemote");
```

Description: Lookup method retrieves the name of Object. Throws naming exception if the binding name is missing from the server. Binding name can be found after deploying the .ear file to the server, at JNDI Tree Page. (Summary of Servers >examplesServer>view JNDI Tree).

Code sample 3:

```
PayTermRemote remote = (PayTermRemote)(ref);
```

Description: Create PayTermRemote object by casting ref object.

Code sample 4:

```
PayTermRef ref = new PayTermRef();
PayTermDesc desc = remote.findPayTermDesc(ref);
```

Description: Invoked Web service method findPayTermDesc as a remote call. Depending on the requirement, the user can vary the binding name and create a different object to invoke the Web service deployed to the server as a remote EJB call using the above code.

Web Service Call as a POJO Call

This section applies to PL/SQL Web service implementations and Java EE Web service implementations.

If an application is a core Java application, it can still call the Web services classes, but as POJO classes. In this case, the Web service classes act as simple Java classes, and there is no marshalling of XML involved, nor a remote call as an EJB.

The PL/SQL provider services need a database connection to call PL/SQL packages. In the case of a Web service call or an EJB call, the service gets the connection from the data source supplied by the Java EE container through resource injection. But in the

case of a Java application, the data source is not available through this mechanism. The connection must be passed to the Web service class before invoking any business methods on it. To achieve this, the caller application must create an instance of the Web service class using the non-default constructor available in the service bean class. An example of the signature of the constructor is below:

```
public PayTermBean(Connection conn,Map<String,String> serviceContext)
```

> **Note:** The bean class is available in the <*appname*>-service-ejb.jar for each Web service generated. For example, if the service name is PayTerm in the service definition XML, the name of the generated bean class will be PayTermBean. This is the class that should be used to call a Web service as a POJO.

In the constructor shown above, the first parameter is for database connection. The second parameter is for the calling application to provide any additional parameters to the bean passed on to the PL/SQL package. When the bean is called as a Web service, an instance of ServiceOpContext class is created by using properties available from an instance of javax.xml.ws.WebServiceContext, available through resource injection. When the bean is called as EJB, an instance of ServiceOpContext is created from the values in an instance of javax.ejb.EJBContext, available through resource injection. But when the bean is called as a POJO, none of these objects is available. Therefore, a map has been added in the constructor so that the calling application can set the required values. If a null object is passed to the constructor for the map, an empty instance of ServiceOpContext is created. If the map contains a key named "user," a Principal object is created with the value of that key, and it is set in the ServiceOpContext object.

Procedure

Complete the following steps.

> **Note:** See the section, "Important Note About this Chapter".

1. Generate the .ear file for Web services and extract the following jar files from it:
   - retail-public-payload-java-beans-base.jar
   - retail-public-payload-java-beans.jar
   - oo-jaxb-bo-converter.jar
   - retail-soa-enabler.jar
   - <*appname*>-service-ejb.jar
2. Include these jar files in the classpath of the Java application that is going to invoke the beans as POJO classes.
3. Write the code to call the bean classes. (Sample code is provided below in this section.)
4. Run the calling class.

Note: The connection must be committed or rolled back by the calling application. Because there is no Java EE container available in this case, the bean cannot start and end a transaction. Therefore, it is the responsibility of the calling application to manage the transaction and the connection. In the following sample code, the calling class is committing the connection in case of a successful response from the bean, and it is rolling back the connection in case of any exception thrown by the Web service. The calling application determines how it wants to handle exceptions.

Sample Code for POJO Invocation

```
public class PayTermService extends TestCase{

       public void testPayTerm(){
               Connection conn = null;
               try{
                     //get the database connection
                     Class.forName("oracle.jdbc.OracleDriver");
                     conn
=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl","stubby","ret
ek");

                    //create map for ServiceOpContext
                    Map<String,String> ctxMap = new HashMap<String, String>();
                    ctxMap.put("user", "user1");

                    //instantiate the web service bean class
                    PayTermBean bean = new PayTermBean(conn,ctxMap);

                    //populate the input object for web service method
                    PayTermRef ref = new PayTermRef();
                    ref.setTerms("terms");
                    ref.setTermsXrefKey("key");

                    //call the web service.here desc is the response object
                    PayTermDesc desc = bean.findPayTermDesc(ref);

                    //print the response object value
                    System.out.println("desc value="+desc.getTerms());

                    //commit the database connection
                    conn.commit();
             }catch(Exception e){
                     e.printStackTrace();
                     try{
                            conn.rollback();
                     }catch(SQLException se){
                             se.printStackTrace();
                     }
             }finally{
                     if(conn !=null){
                             try{
                                    conn.close();
                            }catch(SQLException se){
                                    se.printStackTrace();
                            }
                      }
```

```
                }
            }
}
```

Deploying the Web Service

This section applies to PL/SQL Web service implementations and Java EE Web service implementations.

> **Note:** See the section, "Important Note About this Chapter".

Complete the following steps using the WebLogic Server Administration Console:

1. If necessary, click **Lock and Edit** on the left navigation bar to enable the Install button
2. Navigate to the Deployments page.
3. Click **Install**.

> **Note:** If the service application has already been installed, see "Redeploy the Service Application".

4. The **Locate deployment to install and prepare for deployment** screen is displayed. Follow the instructions to locate the <service-name>.ear file on the WebLogic Server host

   If rib-home is located on a host other than the Oracle WebLogic Server, select Upload Files. On the **Upload a Deployment to the admin server** screen, use the browse button to locate the <service-name>.ear file in the Deployment Archive.
5. Select the igs-service.ear.
6. Click **Next** to move to **Choose targeting style**.
7. Select **Install this deployment as an application**.
8. Click **Next** to move to **Select deployment targets**. Here select the server to which you want the ear file to be deployed.
9. Click **Next** to move to **Optional Settings**. Here in the **Security** section, select the option **Custom Roles and Policies:Use only roles and policies that are defined in the Administration Console**. This is required to be able to attach roles and policies to secure the web services.
10. Click **Next** to move to **Review your choices** and click **Finish**.
11. Select **No, I will review the configuration later**.
12. Click **Finish** to deploy the application.
13. Click **Activate Changes** to commit changes to server.
14. Go to **Deployments** page, select the service application and click on **Start > Servicing all requests** to start the application and change the status to **Active**.

Redeploy the Service Application

If the *<service-name>* application has already been deployed, follow these steps:

1. If the <*service-name*> application is running, select **Stop** and **When Work Completes** or **Force Stop Now**, depending on the environment. The recommended option always is **When Work Completes**.
2. Select **Delete**.
3. The Summary of Deployments should now include the igs-service.
4. Return to "Deploying the Web Service".

Verify the Service Application Installation Using the Administration Console

To verify the Service installations using the Oracle WebLogic Administration Console, follow these steps.

> **Note:** See Oracle WebLogic Server 11g Release 3 (10.3.3) documentation about the Administration console.

1. Navigate to the Deployments screen.
2. Locate the <service-name> on the Summary of Deployments screen.
3. Click **plus sign** next to the ig-service to expand the tree.
4. Locate the Web services section.
5. Click any Web service to move to a Settings for <service name> Service screen.
6. Click the **Testing** tab.
7. Click **plus sign** next to the service name to expand the tree.
8. Click the **Test Clien**t link to move to the WebLogic Test Client screen.
9. Select **Ping Operation**.
10. The test page will show the request message and the response message.

Creating a JDBC Data Source

This section applies to PL/SQL Web service implementations and to Java EE Web service implementations.

To create a JDBC Data Source, follow these steps:

> **Note:** See the section, "Important Note About this Chapter".

1. Log in to the WebLogic administration console. Use the URL, http://<host>:<listen port>/console/login/LoginForm.jsp.
2. Navigate the domain structure tree to Services/JDBC/Data Sources.
3. Click **New** to start creating the new Data Source. Enter the required information:

   Name: Enter any name for the data source.

   JNDI name: This field must be set to jdbc/RetailWebServiceDs. The generated code for the service will use this JNDI name to look up the data source.
4. Select the transaction options for your data source and click **Next.**
5. Enter the database name and user information for the data source. Click **Next**.

6. The screen includes the connection information for your data source. Click **Test Configuration** to ensure the connection information is correct. If it is correct, the following message is displayed: "Connect test succeeded."
7. Click **Next** and select a server to deploy the data source to. This is not necessary at this point if you want to deploy the data source to a server at a later time.
8. Click **Finish** to complete the data source setup. The new data source is displayed on the data sources screen.
9. Click the new data source to view the properties. A default connection pool is created for the data source. Click the **Connection Pool** tab to view the connection pool properties.
10. The generated JDBC connection URL for the data source is displayed. The Oracle URL is formatted as follows: jdbc:oracle:thin:@<hostname>:<port>:<sid>.

    For example: jdbc:oracle:thin:@localhost:1521:orc
11. If the database is a RAC database, the URL should be in the following format

    ```
    jdbc:oracle:thin:@(DESCRIPTION =(ADDRESS_LIST =(ADDRESS = (PROTOCOL = TCP)(HOST
    = <host>)(PORT = <port>))(ADDRESS = (PROTOCOL = TCP)(HOST = <host>)(PORT =
    <port>))(LOAD_BALANCE = yes))(CONNECT_DATA =(SERVICE_NAME = <sid>)))
    ```

    For example:

    ```
    jdbc:oracle:thin:@(DESCRIPTION =(ADDRESS_LIST =(ADDRESS = (PROTOCOL = TCP)(HOST
    = dbhost1.example.com)(PORT = 1521))(ADDRESS = (PROTOCOL = TCP)(HOST =
    dbhost1.example.com)(PORT = 1521))(LOAD_BALANCE = yes))(CONNECT_DATA =(SERVICE_
    NAME = orcl)))
    ```

12. Restart the WebLogic instance to apply the data source changes.

10

Implementation Guidelines For Restful web services

This chapter provides a set of implementation notes that may be helpful when implementing the Oracle Retail Service-Oriented Architecture Enabler (RSE) tool. The information included here is intended to provide guidance on the following topics:

- PL/SQL Provider Service
- Java EE Service Provider
- Deploying the Web service
- Creating a JDBC Data Source

Important Note About this Chapter

The implementation notes in this chapter are intended to provide some guidance in the development and deployment of the Web service layer. This information does not take into account the implementation of the business logic required to complete the application API layer.

The RSE tool and approaches described in this section are complex. A high level of skill and knowledge of the product is required to complete these implementation tasks. Also required is technology specific development of application APIs and the business logic that is needed to complete it.

Any issues that may arise with development tools, development environments, custom APIs, or custom message flows are the responsibility of the customer and not Oracle Retail.

PL/SQL Provider Service Implementation Notes

The distribution (.zip) file includes an .ear file that contains all the generated code for the service; it is ready to be deploy to the application server. The business logic can be implemented in PL/SQL packages in Oracle. The distribution contains the specification and body scripts for the packages called by the deployed service.

To complete implementation, follow these steps:

> **Note:** See Important Note About this Chapter.

1. Create the PL/SQL service provider distribution file using the RSE tool. The output of this process is the .zip file.

Note: See Chapter 4, "User Interface Usage".

2. Extract the <service_name>ProviderImplSpec.sql and <service_name>ProviderImplBody.sql files from the distribution zip file.

   These files are modified to provide a PL/SQL implementation for the service.

3. Extract the <appname>-service.ear file from the distribution zip file. This file is the generated Web service that is deployed.
4. Create the JDBC data source.

Note: See Chapter , "Creating a JDBC Data Source in Glassfish Server".

5. If not already deployed, deploy the Oracle Objects to the appropriate database user.

Note: See the *Oracle Retail Functional Artifact Generator Guide*.

6. Modify the PL/SQL body file for the business logic implementation. The <service_name>ProviderImplBody.sql file contains comments about where to implement logic for each method on the service.
7. Install the modified PL/SQL packages to the database. They are called by the Web service methods.
8. Deploy the <appname>-rest-service.ear file to the Oracle Glassfish Server.

Java EE Service Provider Implementation Notes

The RSE tool creates the appropriate provider restful resources as well as a skeleton implementation layer where the developer implements business logic. All of this is packaged inside the provider distribution archive file.

The Java EE Provider distribution file provides a sample deployable application and all the libraries that can be used to create Web services using retail payloads. The distribution file follows the naming convention of *<appname>*_JavaEEServiceProvider.zip. For example, the distribution file for the RMS application is named rms_JavaEEServiceProvider.zip. The <rms> prefix must be replaced with the name of any other application being developed.

The Web services generated by the RSE tool can be implemented and deployed in a number of ways. This section includes three implementation use cases for reference.

Note: See the section, Important Note About this Chapter.

Use Case 1: Complete the Generator Provided Stub Code Implementation

1. Generate the distribution file using the RSE tool.
2. Extract the <appname>-service-ejb-impl-src.jar file from the zip file.
3. Extract the <appname>-rest-service.ear file from the zip file.

4. Add business logic code where indicated in the Impl java files.
5. Use the java jar command to re-build the <appname>-service-ejb-impl.jar file.
6. Use the jar command to update .ear file with the new implementation jar.
7. Deploy the .ear file to the server.

Use Case 2: Provide a Custom impl jar to the RSE Tool

1. Create custom java classes that implement the <service_name>ServiceProvider interfaces contained in the <appname>-service-provider.jar file.
2. Extract the ServiceProviderImplLookupFactory.properties file from the .ear file.
3. Modify the properties file to point to your implementation classes for the services.
4. Use the jar command to create a jar containing your implementation classes, as well as the modified properties file.
5. Run the RSE tool again and provide the new custom implementation jar file.
6. Extract and deploy the generated .ear file to the server.

Use Case 3: Package the Generated Service Classes in an Existing Application

1. Generate the distribution file using the RSE tool.
2. The service interfaces are provided in the <*appname*>-service-provider.jar file in the distribution file. This jar file should be included in the application classpath.
3. Source code of sample implementations for the service interfaces are provided in the <*appname*>-service-ejb-impl-src.jar file in the distribution file. (If application developers want to use the same classes in their application, they can extract the java files from the jar file and include those in application source code. They also can add their own business logic in the method implementations. If they decide to write their own implementations, they should make sure that the appropriate service interfaces are implemented.)
4. After writing the Web service implementations, the java files should be compiled. The class files can be included in a new jar file or in the same jar file used for the rest of the classes of the application.
5. Modify the ServiceProviderImplLookupFactory.properties file to include appropriate class names of service implementations and include it in application classpath. A recommended approach is to include the properties file in the jar file that contains the service implementation classes.
6. Make sure that the following jar files are included in the application ear file:
   - <*appname*>-service-provider.jar
   - Jar file containing the service implementation classes
   - jersey-client.jar
   - jersey-core.jar
   - jersey-json.jar
   - rms-service-util.jar
   - retail-public-payload-java-beans-base.jar
   - retail-public-payload-java-beans.jar

- retail-soa-enabler.jar

7. Include web-module in the application.xml of the application. The module name should be same as the name of <*appname*>-rest-service.war file.
8. The .ear file is ready for deployment on the server.

Deploying the Web Service

Below are the steps for installing the <service-name>.ear on glassfish server:

1. Download and install Glassfish 3.1.1 server.
2. After installing, go to the glassfish-3.1.1/bin folder and run the below command:

   ./asadmin start-domain -v domain1
3. Access the admin console at URL: http://<host>:4848/
4. For creating an Oracle datasource in glassfish, copy ojdbc6.jar to the path $GLASSFISH_HOME/domains/domain1/lib/ext folder.
5. Stop and start the server again.

   Steps for creating a datasource are listed in the section Creating a JDBC Data Source in Glassfish Server.
6. After creating data source, access the Applications page
7. Click **Deploy**.
8. Browse the <appname>-rest-service.ear file to deploy and click **OK**.

   The deployed application is listed on the Applications page.
9. The URL to test a web service is like this: http://<host>:8080/rms-rest-service/SupplierResource/4/5.
10. In the above example, "rms-rest-service" is the context of web application in ear file, SupplierResource is the name of service, and the numbers are values supplied for path parameters for the web service.

    The above URL makes a call to the GET method of the web service.
11. For testing all other operations you can install SOAP-UI. And provide the URL of the WADL to create test cases. A sample URL of the WADL is http://<host>:8080/rms-rest-service/application.wadl
12. It will show all the operations that are available for a web service. You can enter request xml and execute the web service method.

Creating a JDBC Data Source in Glassfish Server

To create a JDBC Data Source in Glassfish Server, follow these steps:

1. Select **Resources > JDBC > JDBC Connection Pools**.
2. Click **New**.
3. In the Pool Namefield, enter **OraclePool**.
4. Select Resource Type as **javax.sql.DataSource**.
5. Select driver vendor as **Oracle**.
6. In Additional Properties section, enter the following details:

- user: **<*database user name*>**
- DatabaseName: **<*sid*>**
- password: **<*database password*>**
- URL: **jdbc:oracle:thin:@<*host*>:<*port*>:<*sid*>**

7. Save the connection pool.
8. Select **Resources > JDBC > JDBC Resources**.
9. Click **New**.
10. Enter JNDI Name: jdbc/RetailWebServiceDs
11. Select Pool Name as the one that was just created.
12. Click **OK**.
13. After saving the changes, restart the server.

11

Web Services Security Setup Guidelines

There are numerous ways to build or implement secured service to protect the SOA infrastructure against attack. Standards allow policies to be applied to SOA, thus allowing controlled usage and monitoring and provide security ramifications in enterprise integration. Standards such as WS-Security, SAML, WS-Trust, WS-Secure Conversation and WS-SecurityPolicy focus on the security and identity management aspects of SOA implementations that use web services.

The WS-* architecture is a set of standards-based protocols designed to secure Web service communication. WebLogic Web Services use WS-Policy files to enable a destination endpoint to describe and advertise its Web Service reliable messaging capabilities and requirements. The WS-Policy specification provides a general purpose model and syntax to describe and communicate the policies of a Web service.

These WS-Policy files are XML files that describe features such as the version of the supported WS-ReliableMessaging specification, the source endpoint's retransmission interval, the destination endpoint's acknowledgment interval, and so on.

The web services exposed by Oracle retail applications are used as service providers in Retail Service Backbone (RSB) architecture. Please refer to RSB documentation for more details about RSB architecture.The Oracle Retail application services are used as edge application services in RSB and they are consumed by Web services through the OSB layer. When used with RSB, the Oracle Retail application services are not consumed directly, instead the consumers invoke OSB services which in turn invoke the Oracle retail application services. Due to these requirements, Oracle Retail application services need to be secured with WebLogic Web service polices, which are interoperable with OWSM policies. Following is the list of WebLogic Web service policies that are currently supported for securing application services

1. Username token over SSL: The following WebLogic policy is used for username token over SSL, it is also referred to as PolicyA in RSB documentation:

   Wssp1.2-2007-Https-UsernameToken-Plain.xml:

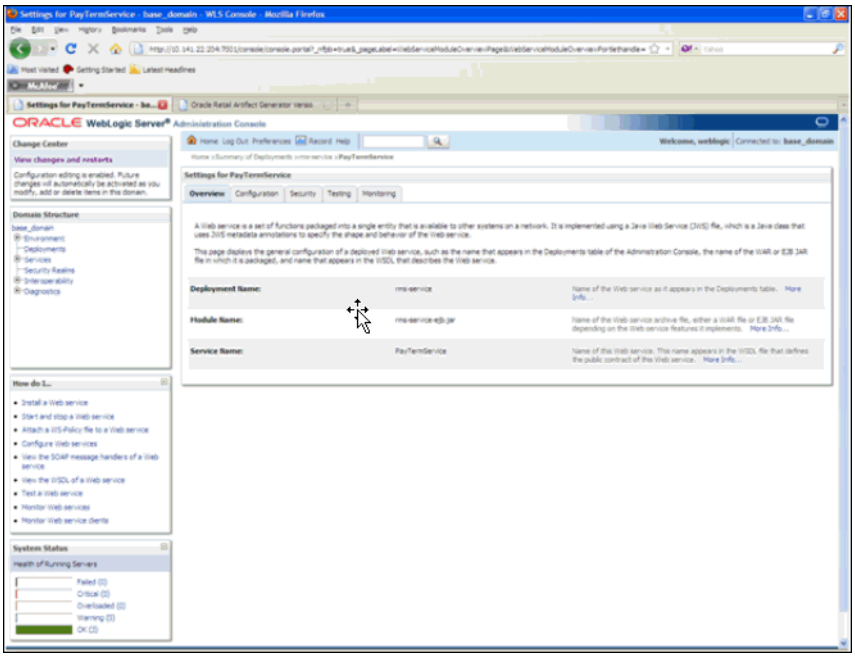
2. Username token with Message Protection: Following is the set of policies which are used to secure services with username token and message protection. This is also referred to as PolicyB in RSB documentation:

   Wssp1.2-2007-Wss1.1-UsernameToken-Plain-EncryptedKey-Basic128.xml

   Wssp1.2-2007-SignBody.xml

   Wssp1.2-2007-EncryptBody.xml

This document doesn't go into the detailed steps for securing Web services. The detailed step-by-step instructions are provided in RSB Security Guide. Refer to that document for more details.



Client-Side Setup

Web services can be invoked from Java clients as well as PL/SQL clients. This section describes the configuration for invoking a secured Web service from both clients.

Java Client Setup

Client code for calling Web services can be generated using the Java consumer option of the retail-soa-enabler-gui tool. The generated zip file contains all the jar files required for the classpath of the application that calls the Web service. To run the client, follow the steps required to run Java consumer.

The following is sample code for calling a secured Web service.

> **Note:** The code below is sample code for invoking the PayTerm service. When you generate Java consumer for a Web service, the generated jar file will contain classes specific to that Web service. Use the appropriate classes in the client code. Service namespace and WSDL location also should be changed appropriately.

```
package com.oracle.retail.rms.client;

import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import javax.xml.namespace.QName;
import javax.xml.ws.BindingProvider;
```

```
import com.oracle.retail.integration.base.bo.paytermdesc.v1.PayTermDesc;
import com.oracle.retail.integration.base.bo.paytermref.v1.PayTermRef;
import
com.oracle.retail.rms.integration.services.paytermservice.v1.PayTermPortType;
import
com.oracle.retail.rms.integration.services.paytermservice.v1.PayTermService;

import weblogic.wsee.security.unt.ClientUNTCredentialProvider;
import weblogic.xml.crypto.wss.WSSecurityContext;
import weblogic.xml.crypto.wss.provider.CredentialProvider;

import junit.framework.TestCase;


public class PayTermClient extends TestCase{
       public void testFindPayTerm(){
              try{
                      //qName is namespace of the service
                      QName qName = new
QName("http://www.oracle.com/retail/rms/integration/services/PayTermService/v1","P
ayTermService");

                      // url is the URL of the WSDL of the web service
                      URL url = new
URL("http://10.141.26.93:7001/PayTermBean/PayTermService?WSDL");

                     //create an instance of the web service
                     PayTermService service = new PayTermService(url,qName);
                     PayTermPortType port =   service.getPayTermPort();

                     //set the security credentials in the service context
                     List credProviders = new ArrayList();
                     CredentialProvider cp = new
ClientUNTCredentialProvider("rmsuser","rmsuser1");
                     credProviders.add(cp);
                     Map<String, Object> rc =
((BindingProvider)port).getRequestContext();
                     rc.put(WSSecurityContext.CREDENTIAL_PROVIDER_LIST,
credProviders);

                      //populate the service method input object
                    PayTermRef ref = new PayTermRef();
                    ref.setTerms("terms");
                    ref.setTermsXrefKey("key");

                    //call the web service.here desc is the response object
                    PayTermDesc desc =  port.findPayTermDesc(ref);

                    System.out.println("desc="+desc);
             }catch(Exception e){
                   e.printStackTrace();
             }
      }
}
```

PL/SQL Client Setup

Client code for calling Web services can be generated using the PL/SQL consumer option of the retail-soa-enabler-gui tool. The generated zip file contains all the jar files

and PL/SQL code required to invoke the web service from PL/SQL. To run the client, follow the steps required to run PL/SQL consumer.

The following is a sample PL/SQL procedure for calling a secured Web service.

> **Note:** The code below is sample code for invoking the PayTerm service. When you generate PL/SQL consumer for a Web service, the generated jar file will contain classes specific to that Web service. Use the appropriate classes in the client code. Service namespace and WSDL location should also be changed appropriately.

```
create or replace
PROCEDURE wstestClient IS

ref "OBJ_PAYTERMREF" := null ;
objdesc "OBJ_PAYTERMDESC" := null;
wsm varchar2(100);config varchar2(32000);
BEGIN
wsm := 'oracle.webservices.dii.interceptor.pipeline.port.config';
   config :='<port-info>
   <runtime enabled="security">
   <security>
   <outbound>
   <username-token name="" password=""/>
   </outbound>
   </security>
   </runtime>
   </port-info>' ;

 PayTermServiceConsumer.setProperty(wsm, config);

PayTermServiceConsumer.setEndpoint('http://10.141.22.204:7001/PayTermBean/PayTermS
ervice');
PayTermServiceConsumer.setUsername('<RMS username>');
PayTermServiceConsumer.setPassword('<RMS password>');
ref := "OBJ_PAYTERMREF"('x','t',null,null,null);
dbms_output.PUT_LINE(PayTermServiceConsumer.getEndPoint());
dbms_output.PUT_LINE(PayTermServiceConsumer.ping('TestMessage'));
objdesc := PayTermServiceConsumer.findPayTermDesc(ref) ;
dbms_output.PUT_LINE('Done.');
EXCEPTION
   WHEN OTHERS THEN
    dbms_output.PUT_LINE(SQLCODE);
   dbms_output.PUT_LINE(SQLERRM);
END;
```

A

Appendix: Installer Screens

This appendix provides step-by-step instructions for installing the Oracle Retail Service-Oriented Architecture Enabler tool as a Web application in Oracle WebLogic.

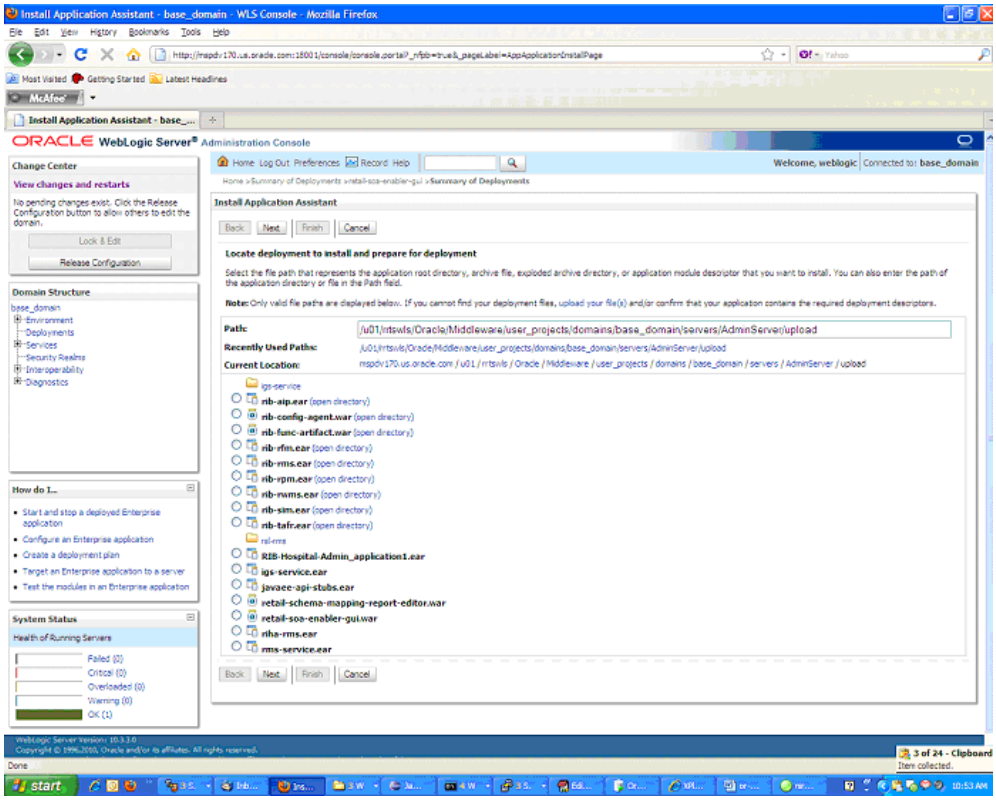
Installation as a Web Application in Oracle WebLogic

To install the Oracle Retail Service-Oriented Architecture Enabler tool as a Web application in Oracle WebLogic, complete these steps.

Deploy the Retail SOA Enabler Application

Using the WebLogic Server Administration Console, complete the following steps:

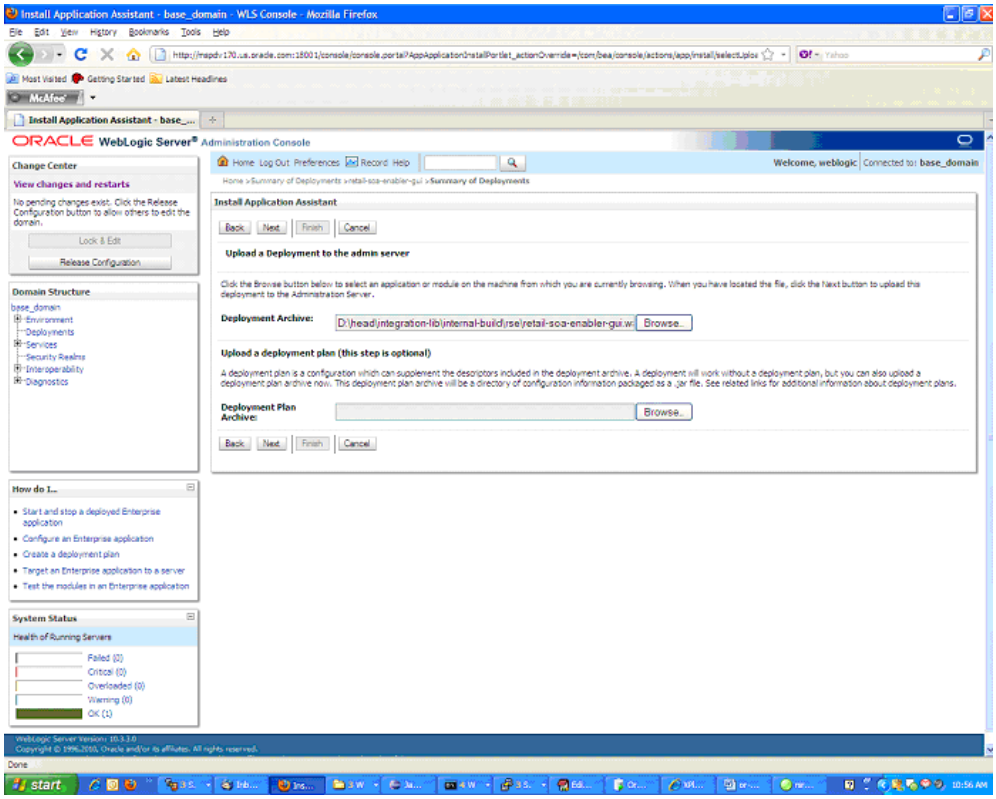
1. Navigate to the Deployments page:
2. In the left navigation bar, click **Lock & Edit**. Click **Install**.



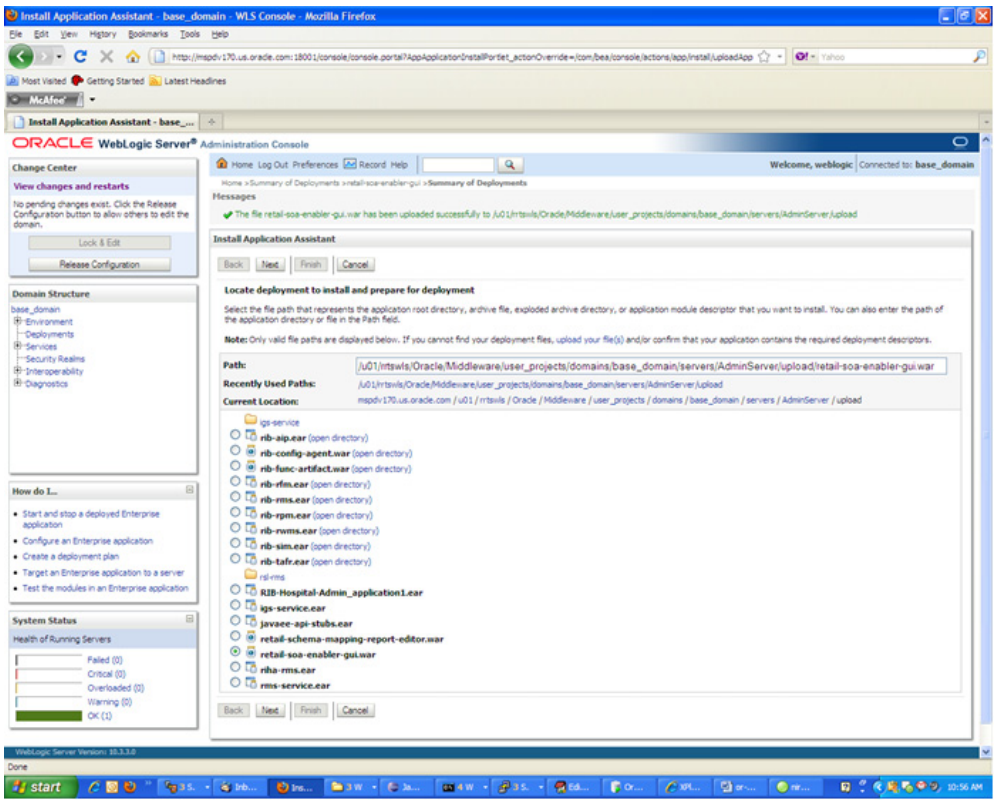
Note: If the application has already been installed, see the section, "Redeploy the Application."

The **Locate deployment to install and prepare for deployment page** is displayed. Follow the instructions to locate the retail-soa-enabler-gui.war file.

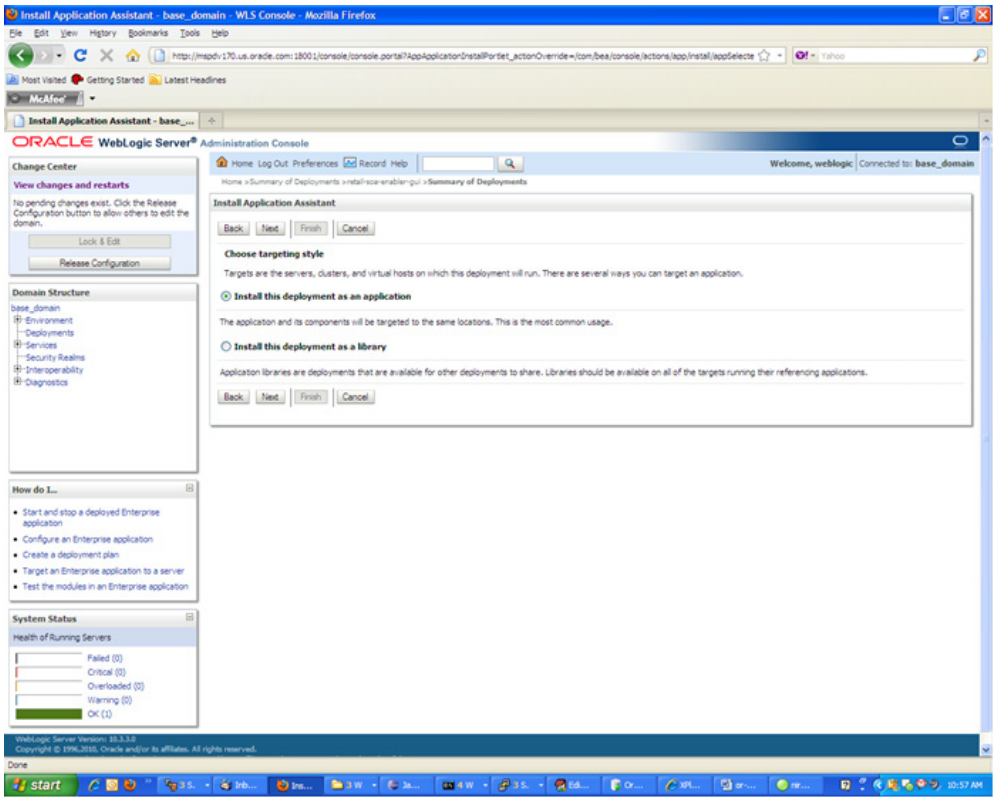
3. Select **Upload Files**.
4. On the **Upload a Deployment to the admin server** page, use the **Browse** button to locate the retail-soa-enabler-gui.war file in the Deployment Archive.



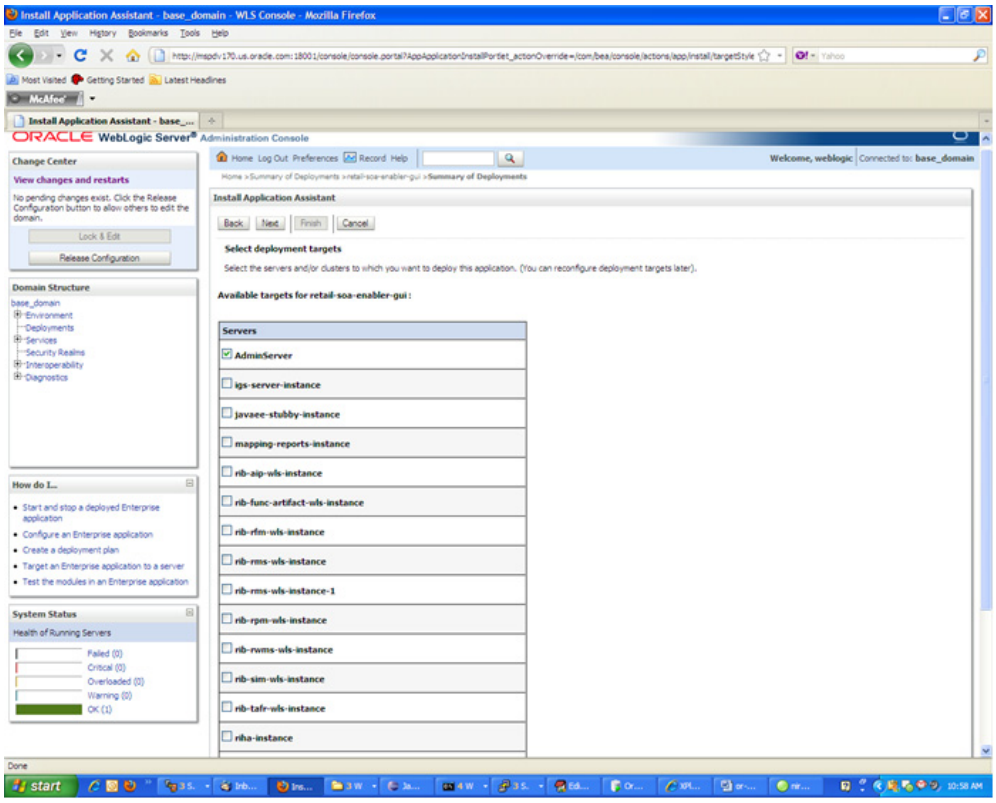
5. Select the retail-soa-enabler-gui.war.



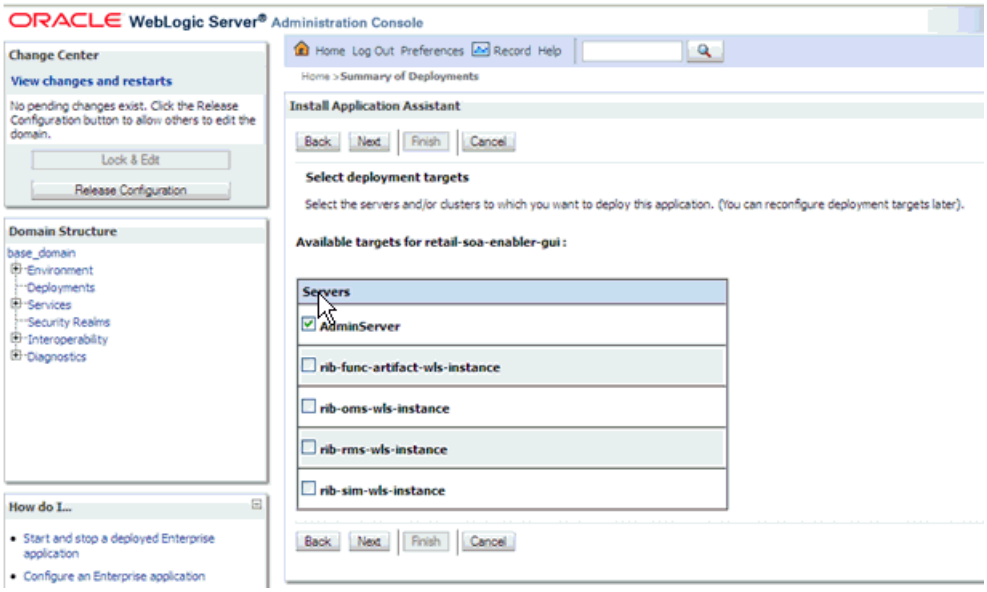
6. Click **Next** and move to **Choose targeting style**.



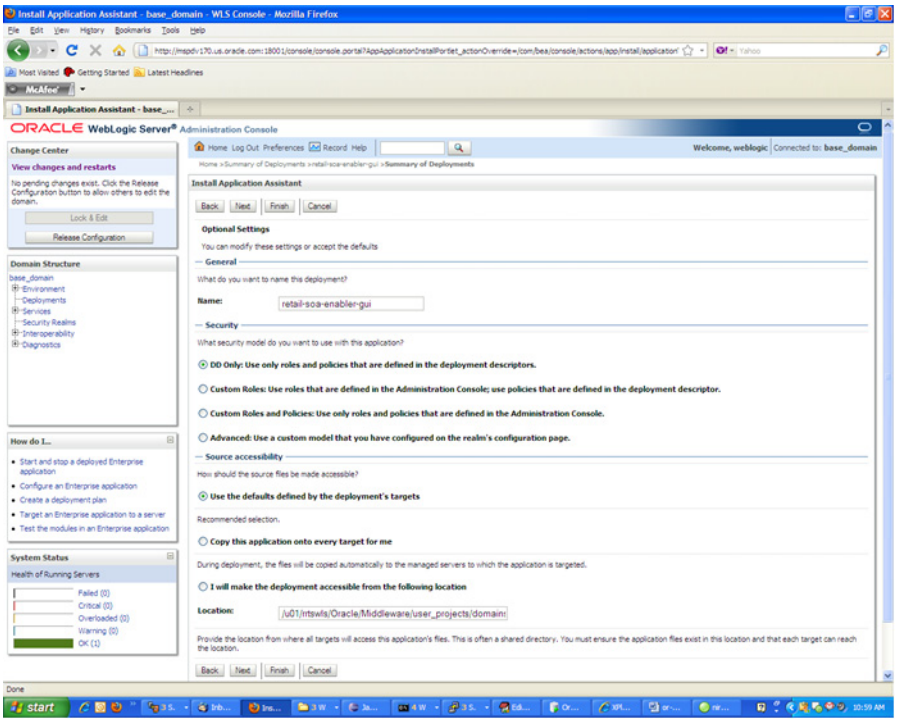
7. Select **Install this deployment as an application**.



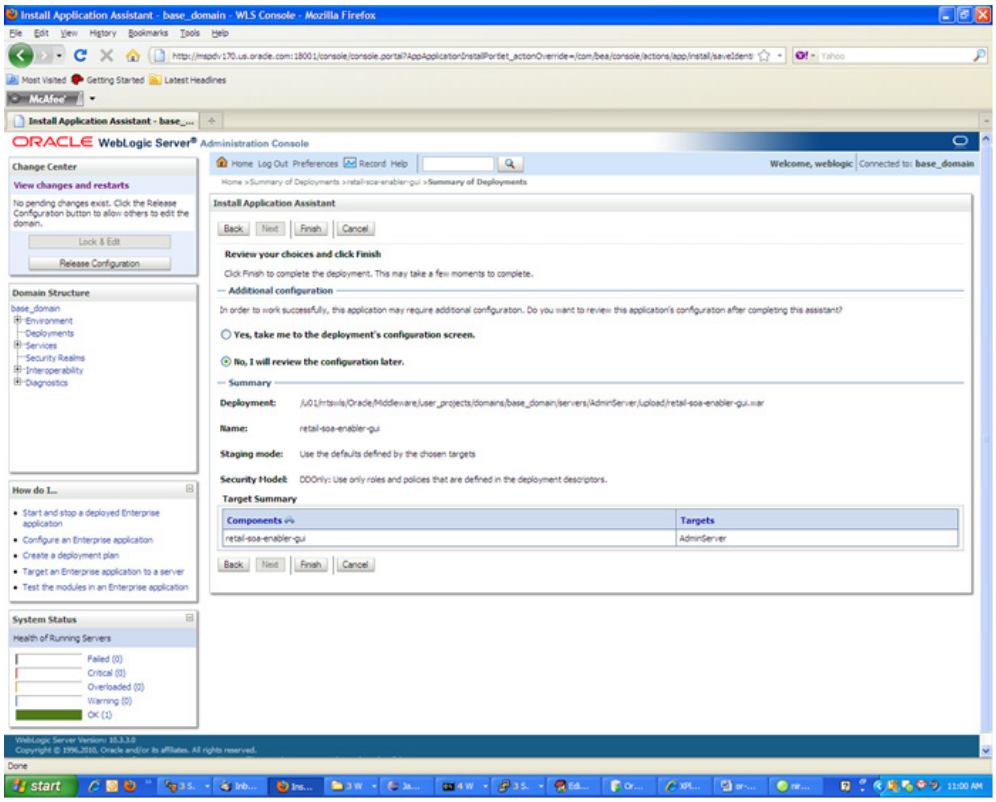
8. Select Deployment Target.



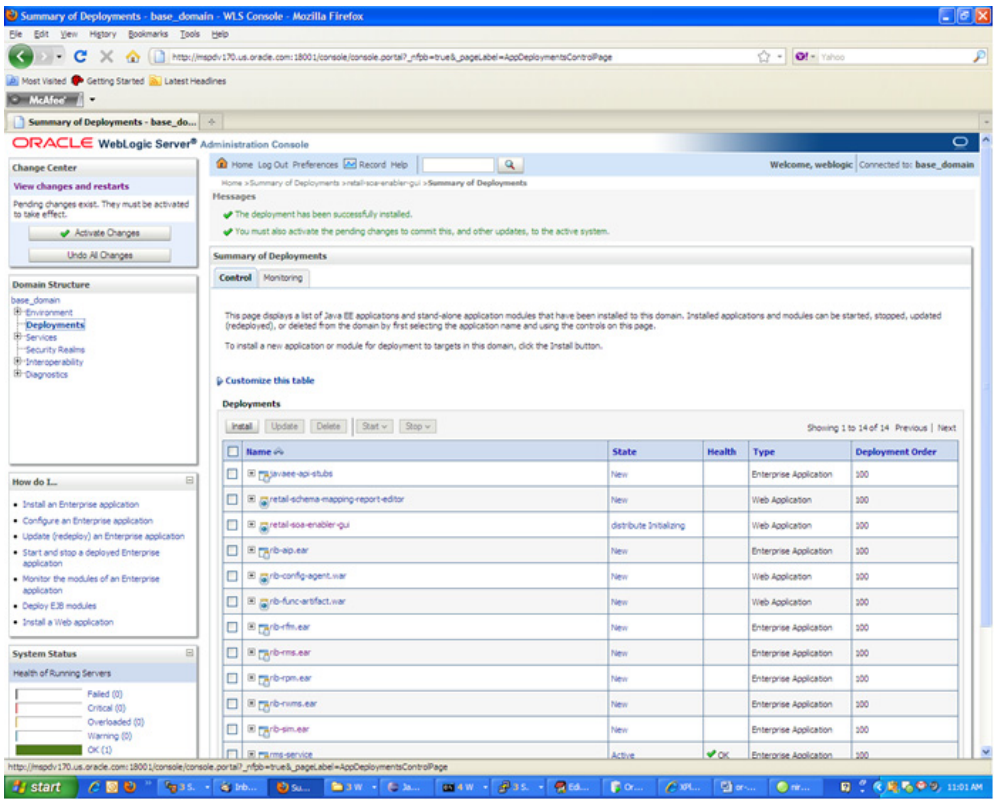
9. Click **Next** to select security options. Here select the option **DD Only: Use only roles and policies that are defined in the deployment descriptors**.



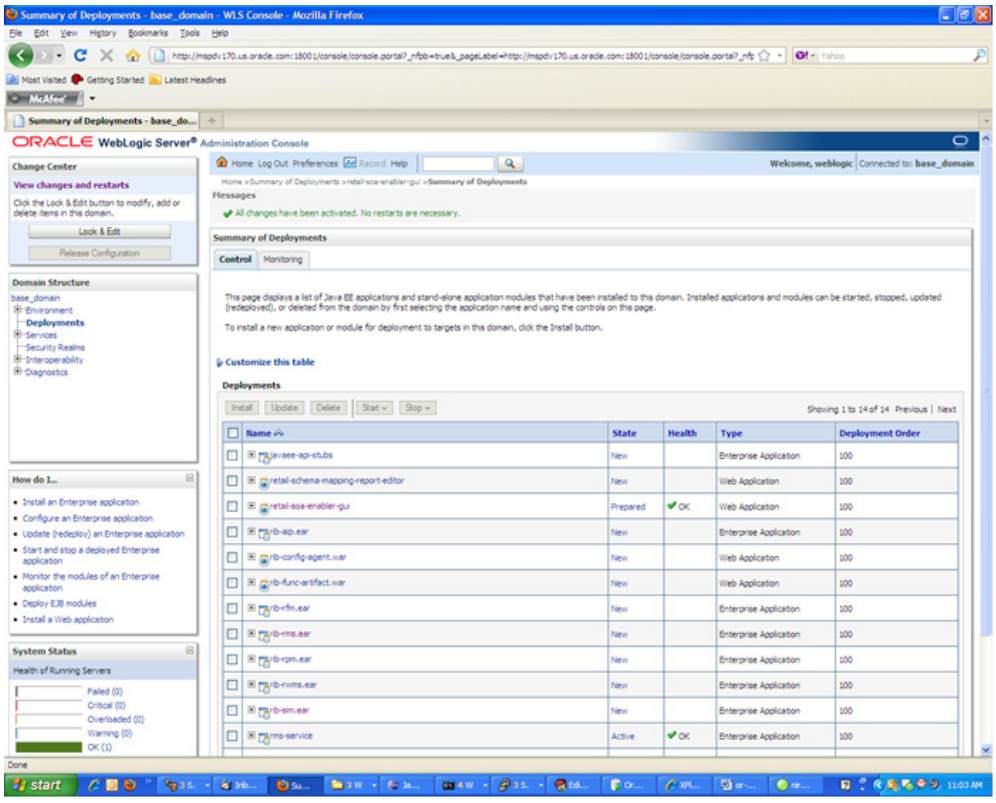
10. Click **Next** to review your choices. Click **Finish**.



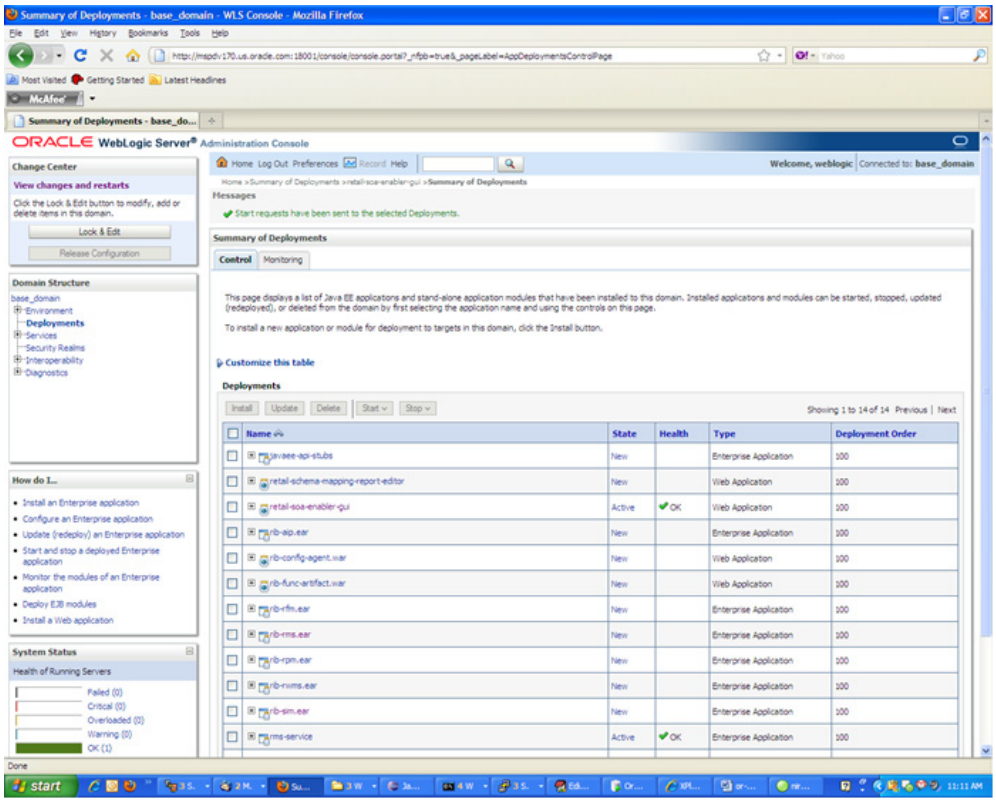
11. Select **No, I will review the configuration later**.
12. Click **Finish** to deploy the application.



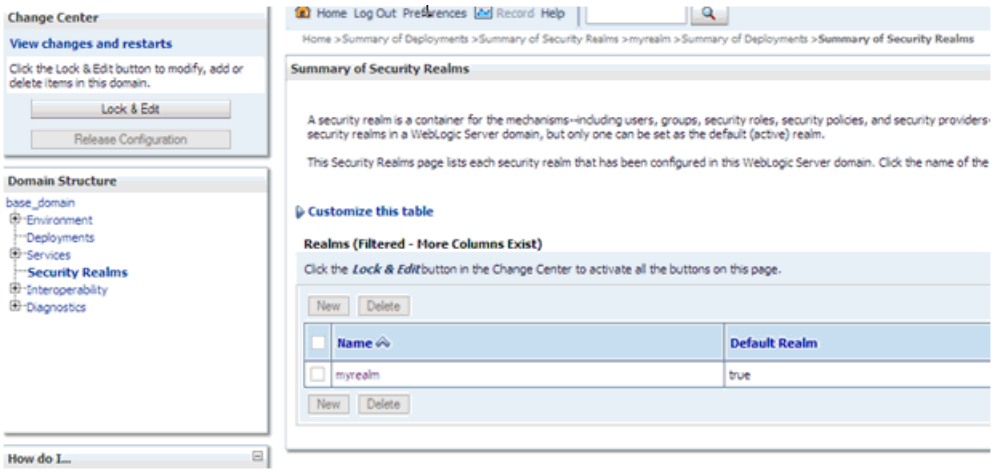
13. Click **Activate Changes** to finish the deployment.



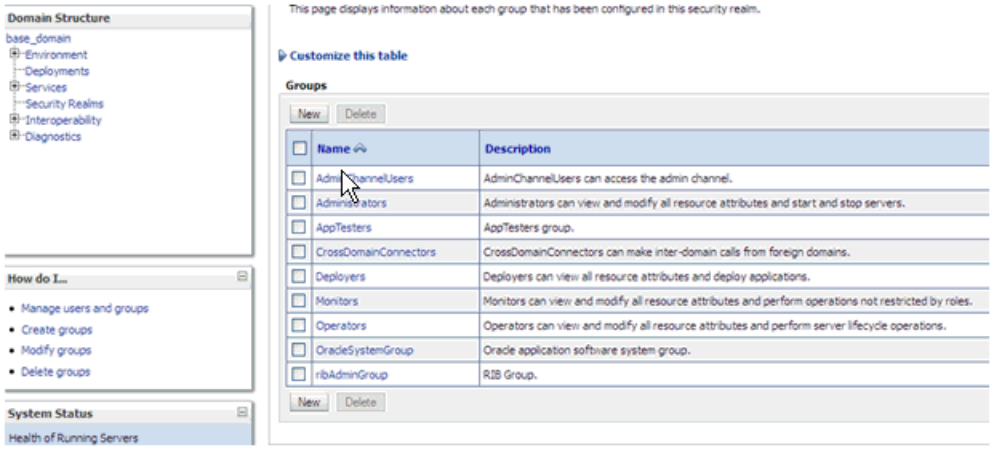
14. Select the retail-soa-enabler-gui application. Click **Start > Servicing All Requests.**



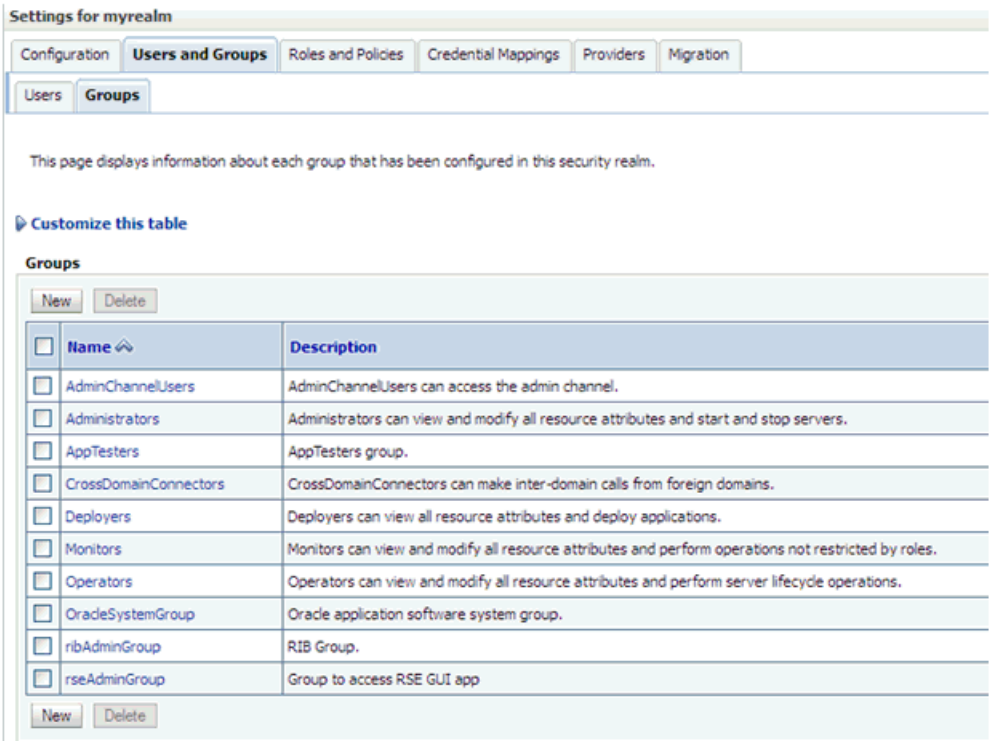
15. Next step is to add group and user required to acess RSE GUI application. To create the group click on **Security Realms**.



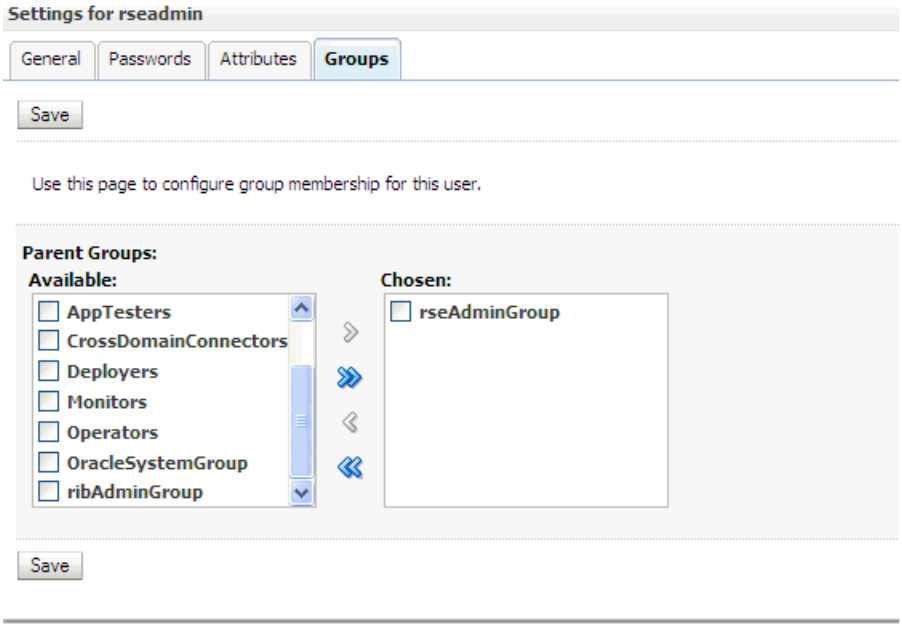
16. Click on the default realm name and go to **Users and Groups** tab, and go to Groups tab.



17. Click on **New** button. In the next page, enter the name of group as rseAdminGroup. Enter description for the group.
18. Click **OK** button. The new group gets added.
19. Now go to the **Users** tab of security realm.
20. Click on **New** button to create a new user. In the next page, enter username and password for the new user to be created.
21. Click **OK** button.The new user gets added.



22. Now click on the new user and go to Groups tab of that user.
23. Select the group rseAdminGroup from the Available window and move it to Chosen window.



24. Click **Save** button. It will add the newly created user to the group rseAdminGroup.

    This completes the security setup for RSE GUI application. Now if you go to the RSE GUI login page, you should be able to login using the new user.

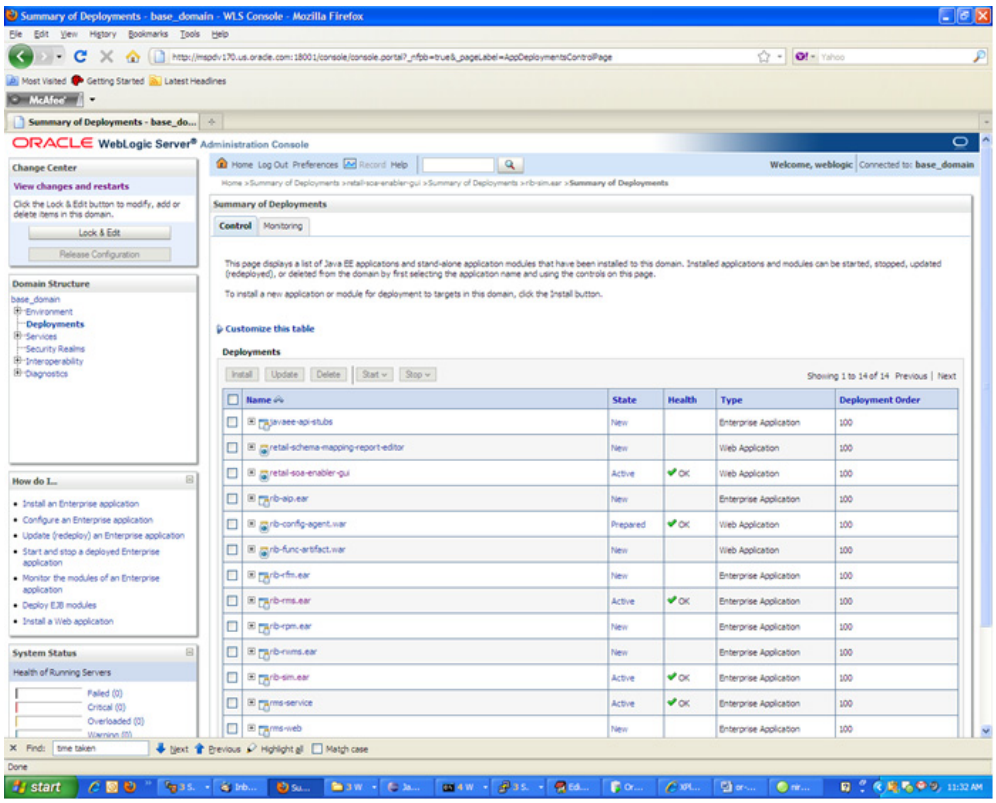
Creating the rseAdminGroup

To create the rseAdminGroup, do the following:

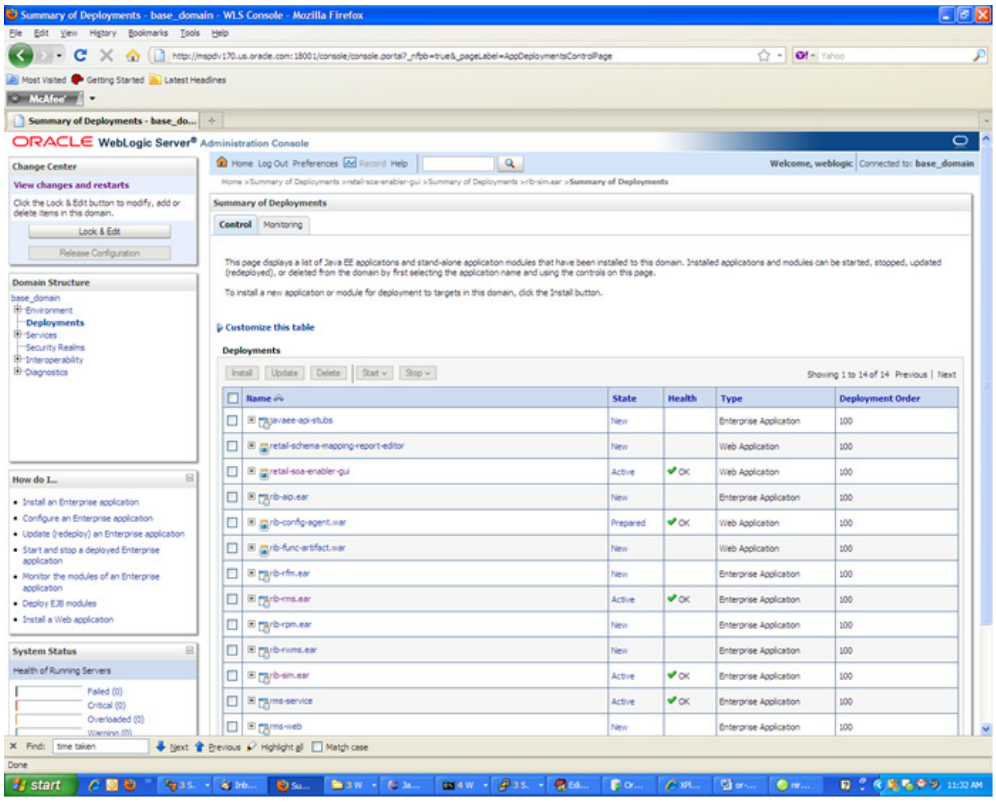
1. In WebLogic, click **Security Realms**.
2. Click on **myrealm** and then **Users** and **Groups**.
3. Click on groups and then **New**.
4. Enter **rseAdminGroup** in the name field, leaving the other fields at default.
5. Click **OK**.
6. Add at least one user to the rseAdminGroup group.

Verify the Retail SOA Enabler Web Application

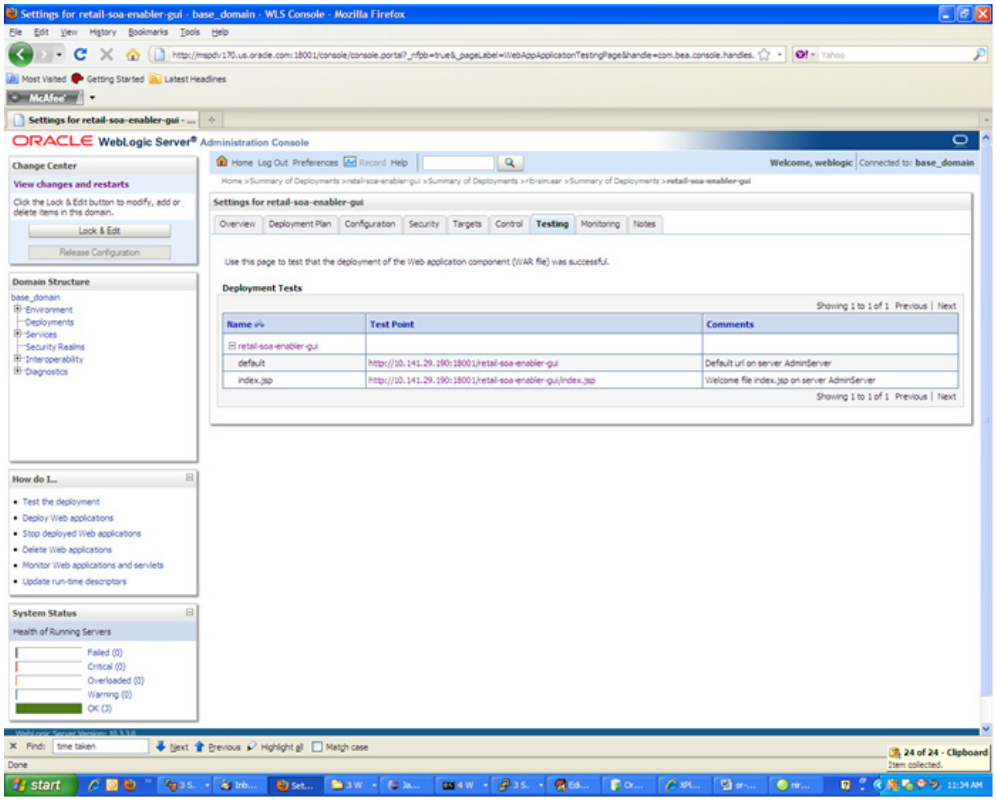
1. Navigate to the Deployments page.



2. On the Summary of Deployments screen, locate retail-soa-enabler-gui.



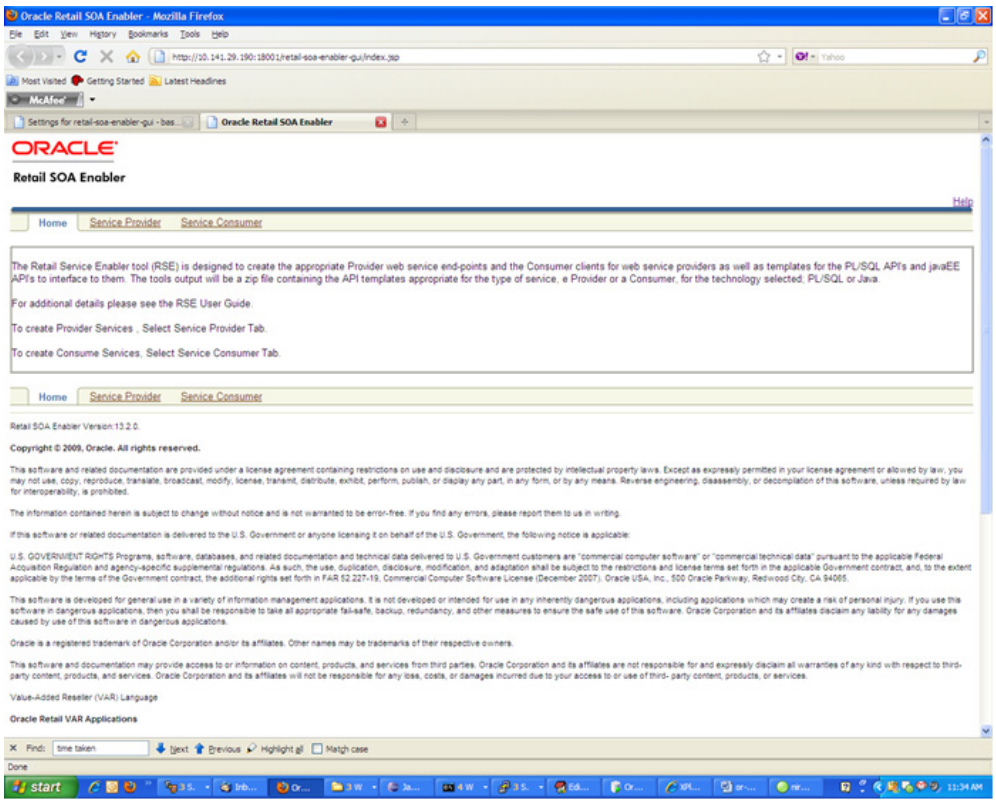
3. Click **retail-soa-enabler-gui** to view settings for the retail-soa-enabler-gui.
4. Select the **Testing** tab.



5. Click the index.jsp URL in the Test Point.
6. The URL should open to the login screen for the Retail Service-Oriented Architecture Enabler Home page.

7. Enter the credentials created in Creating the rseAdminGroup section.

   The RSE home page is displayed.

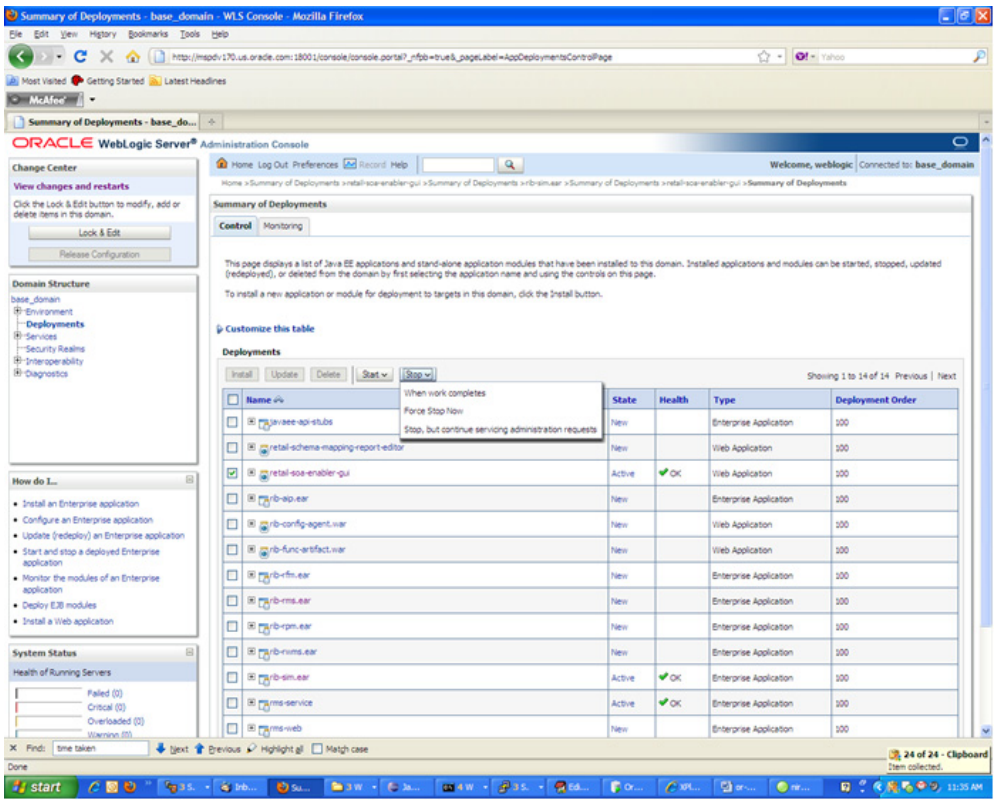


8. The installation is complete. See Chapter 4, "User Interface Usage."

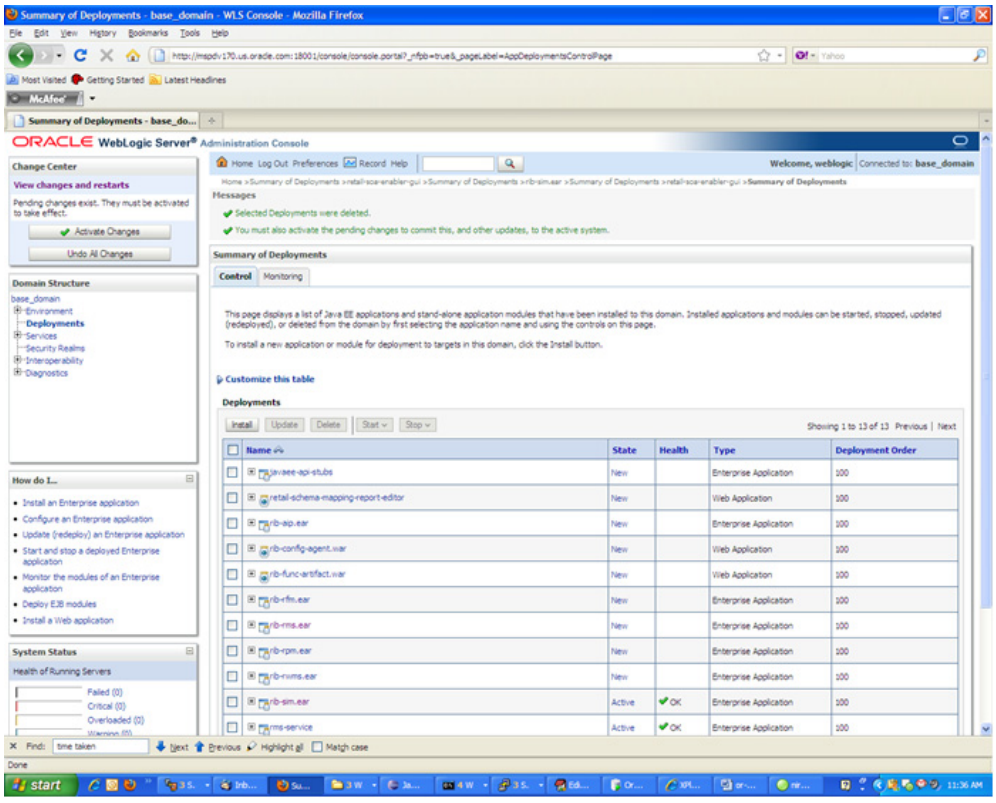
Redeploy the Application

If the retail-soa-enabler-gui application has already been deployed, follow these steps:

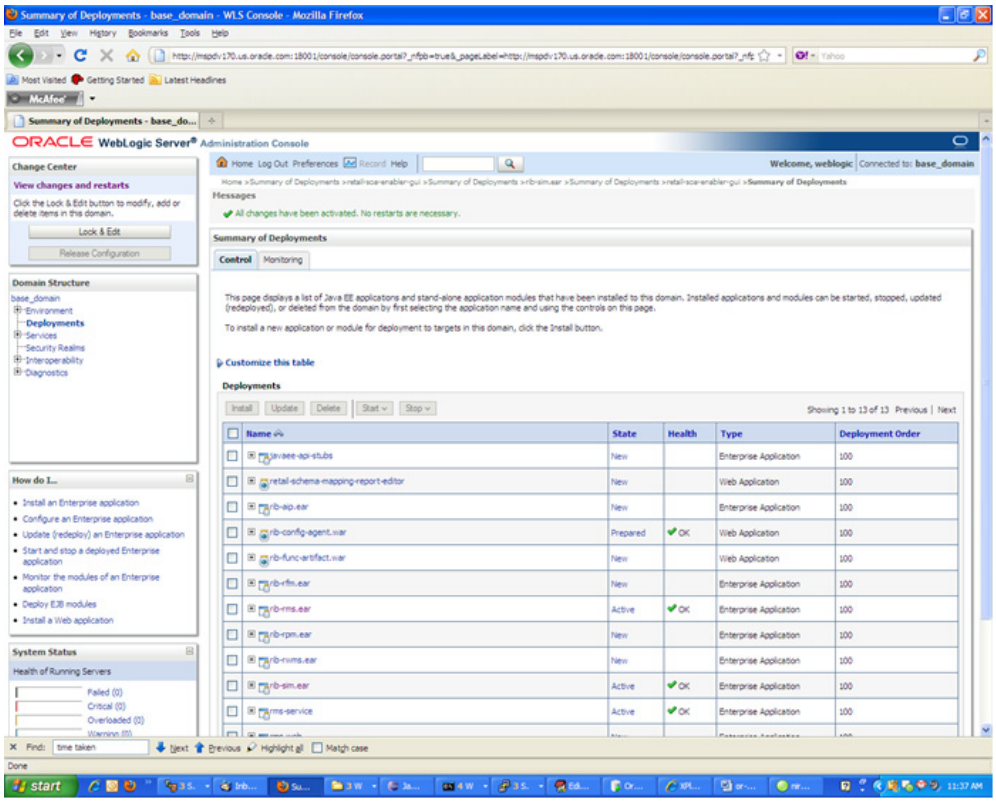
1. If the retail-soa-enabler-gui application is running, select **Stop** and **When Work Completes** or **Force Stop Now**, depending on the environment. The recommended option always is **When Work Completes**.



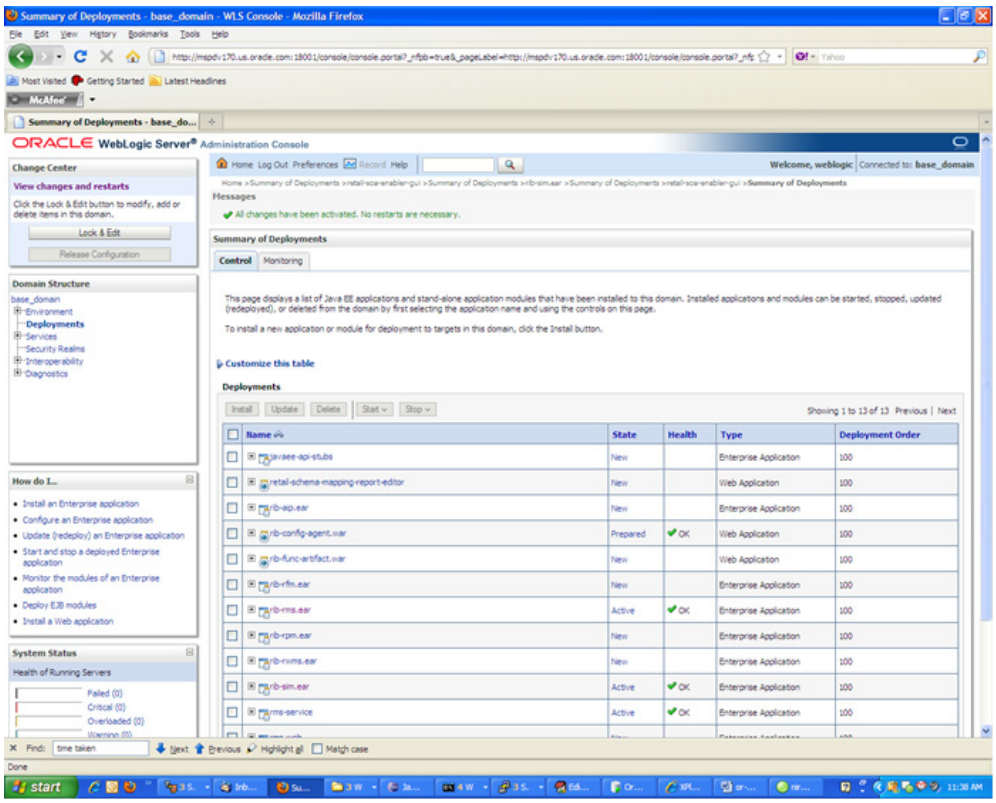
2. Click **Lock & Edit.** Click **Delete**.



3. Click **Activate Changes.**



4. The retail-soa-enabler-gui should now not show on the Summary of Deployment screen.


Summary of Deployments - base_domain - WLS Console - Mozilla Firefox
ORACLE WebLogic Server® Administration Console
Change Center
View changes and restarts
Lock & Edit
Release Configuration
Domain Structure
base_domain
Environment
Deployments
Services
Security Realms
Interoperability
Diagnostics
How do I...
Install an Enterprise application
Configure an Enterprise application
Update (redeploy) an Enterprise application
Start and stop a deployed Enterprise application
Monitor the modules of an Enterprise application
Deploy EJB modules
Install a Web application
System Status
Health of Running Servers
Failed (0)
Critical (0)
Overloaded (0)
Welcome, weblogic
Connected to: base_domain
Messages
All changes have been activated. No restarts are necessary.
Summary of Deployments
Control
Monitoring
This page displays a list of Java EE applications and stand-alone application modules that have been installed to this domain. Installed applications and modules can be started, stopped, updated (redeployed), or deleted from the domain by first selecting the application name and using the controls on this page.
To install a new application or module for deployment to targets in this domain, click the Install button.
Customize this table
Deployments
Install
Update
Delete
Start
Stop
Showing 1 to 13 of 13 Previous | Next
Name
State
Health
Type
Deployment Order
javaee-api-stubs
retail-schema-mapping-report-editor
rib-aip.ear
rib-config-agent.war
rib-func-artifact.war
rib-rfm.ear
rib-rms.ear
rib-rpm.ear
rib-rwms.ear
rib-sim.ear
rms-service
Enterprise Application
Web Application
New
Prepared
Active
OK
100
Find:
time taken
Next
Previous
Highlight all
Match case
Done
start

B

Appendix: Sample ServiceProviderDefLibrary.xml

The sample below can be used as an initial template.

ServiceProviderDefLibrary.xml

```
<serviceProviderDefLibrary appName="rms"
xmlns="http://www.oracle.com/retail/integration/services/serviceProviderDefLibrary
/v1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

         <service name="Supplier"><!-- Noun, don't put suffix Service -->
                  <documentation />
                  <operation name="create"><!-- Verb -->
                           <documentation>Create a new
SupplierDesc.</documentation>
                           <input type="SupplierDesc"><!-- Existing BO -->
                                    <documentation>
                                           Input SupplierDesc to create.
                                    </documentation>
                           </input>
                           <output type="SupplierRef">
                                    <documentation>
                                             Return the SupplierRef for the newly
created
SupplierDesc.
                                    </documentation>
                           </output>
                           <fault faultType="IllegalArgumentWSFaultException">
                                    <documentation>
                                             Throw this exception when it is
"soap:Client" side
message problem.
                                    </documentation>
                           </fault>
                           <fault
                                 faultType="EntityAlreadyExistsWSFaultException">
                                   <documentation>
                                           Throw this exception when the object
already exist.
                                   </documentation>
                           </fault>
                           <fault faultType="IllegalStateWSFaultException">
                                   <documentation>
                                           Throw this exception when there is
```

```
unknown
"soap:Server" side problem.
                                    </documentation>
                            </fault>
                    </operation>
            <operation name="createSupSiteUsing"><!-- Verb -->
                                <documentation>Create a new
SupplierSite.</documentation>
                                <input type="SupplierDesc"><!-- Existing BO  -->
                                        <documentation>
                                                Input SupplierDesc to create.
                                        </documentation>
                                </input>
                                <output type="SupplierRef">
                                        <documentation>
                                                Return the SupplierRef for the
newly created
SupplierDesc.
                                        </documentation>
                                </output>
                                <fault faultType="IllegalArgumentWSFaultException">
                                        <documentation>
                                                Throw this exception when it is
"soap:Client" side
message problem.
                                        </documentation>
                                </fault>
                                <fault
                                  faultType="EntityAlreadyExistsWSFaultException">
                                        <documentation>
                                                Throw this exception when the
object already exist.
                                        </documentation>
                                </fault>
                                <fault faultType="IllegalStateWSFaultException">
                                        <documentation>
                                                Throw this exception when there
is unknown
"soap:Server" side problem.
                                        </documentation>
                                </fault>
                    </operation>
            <operation name="createSupSiteAddrUsing"><!-- Verb -->
                                <documentation>Create a new
SupplierSite.</documentation>
                                <input type="SupplierDesc"><!-- Existing BO -->
                                        <documentation>
                                                Input SupplierDesc to create.
                                        </documentation>
                                </input>
                                <output type="SupplierRef">
                                        <documentation>
                                                Return the SupplierRef for the
newly created
SupplierDesc.
                                        </documentation>
                                </output>
                                <fault faultType="IllegalArgumentWSFaultException">
                                        <documentation>
                                                Throw this exception when it is
```

```
"soap:Client" side
message problem.
                                        </documentation>
                              </fault>
                              <fault
                                   faultType="EntityAlreadyExistsWSFaultException">
                                        <documentation>
                                                 Throw this exception when the
object already exist.
                                        </documentation>
                              </fault>
                              <fault faultType="IllegalStateWSFaultException">
                                        <documentation>
                                                Throw this exception when there is
unknown
"soap:Server" side problem.
                                        </documentation>
                              </fault>
                    </operation>
                    <operation name="update">
                              <input type="SupplierDesc" />
                              <output type="SupplierDesc" />
                              <fault faultType="IllegalArgumentWSFaultException" />
                              <fault
                                        faultType="EntityNotFoundWSFaultException"
/>
                              <fault faultType="IllegalStateWSFaultException" />
                    </operation>
    <operation name="updateSupSiteUsing">
                              <input type="SupplierDesc" />
                              <output type="SupplierDesc" />
                              <fault faultType="IllegalArgumentWSFaultException" />
                              <fault
                                        faultType="EntityNotFoundWSFaultException"
/>
                              <fault faultType="IllegalStateWSFaultException" />
                    </operation>
    <operation name="updateSupSiteOrgUnitUsing">
                              <input type="SupplierDesc" />
                              <output type="SupplierDesc" />
                              <fault faultType="IllegalArgumentWSFaultException" />
                              <fault
                                        faultType="EntityNotFoundWSFaultException"
/>
                              <fault faultType="IllegalStateWSFaultException" />
                    </operation>
     <operation name="updateSupSiteAddrUsing">
                              <input type="SupplierDesc" />
                              <output type="SupplierDesc" />
                              <fault faultType="IllegalArgumentWSFaultException" />
                              <fault
                                       faultType="EntityNotFoundWSFaultException"
/>
                              <fault faultType="IllegalStateWSFaultException" />
                    </operation>
                    <operation name="find" suffix="outputType">
                              <input type="SupplierRef" />
                              <output type="SupplierDesc" />
                              <fault faultType="IllegalArgumentWSFaultException" />
                              <fault
```

```
                                        faultType="EntityNotFoundWSFaultException"
/>
                        <fault faultType="IllegalStateWSFaultException" />
                    </operation>
                    <operation name="delete">
                        <input type="SupplierRef" />
                        <output type="SupplierRef" />
                        <fault faultType="IllegalArgumentWSFaultException" />
                        <fault
                                faultType="EntityNotFoundWSFaultException"
/>
                        <fault faultType="IllegalStateWSFaultException" />
                    </operation>
                    <operation name="create">

                        <input type="SupplierCollectionDesc" />
                        <output type="SupplierCollectionRef" />
                        <fault faultType="IllegalArgumentWSFaultException" />
                       <fault faultType="EntityAlreadyExistsWSFaultException"
/>
                        <fault faultType="IllegalStateWSFaultException" />

                    </operation>
                    <operation name="update">
                        <input type="SupplierCollectionDesc" />
                        <output type="SupplierCollectionDesc" />
                        <fault faultType="IllegalArgumentWSFaultException" />
                        <fault
                                faultType="EntityNotFoundWSFaultException"
/>
                        <fault faultType="IllegalStateWSFaultException" />
                    </operation>
                    <operation name="find" suffix="outputType">
                        <input type="SupplierCollectionRef" />
                        <output type="SupplierCollectionDesc" />
                        <fault faultType="IllegalArgumentWSFaultException" />
                        <fault
                                faultType="EntityNotFoundWSFaultException"
/>
                        <fault faultType="IllegalStateWSFaultException" />
                    </operation>
                    <operation name="delete">
                        <input type="SupplierCollectionRef" />
                        <output type="SupplierCollectionRef" />
                        <fault faultType="IllegalArgumentWSFaultException" />
                        <fault
                                faultType="EntityNotFoundWSFaultException"
/>
                        <fault faultType="IllegalStateWSFaultException" />
                    </operation>
            </service>
</serviceProviderDefLibrary>
```

C

Appendix: Creating a JDBC Data Source

This section describes the steps required to create a JDBC data source.

Procedure

To create a JDBC data source, complete the following steps.

1. Log in to the WebLogic administration console. Use this URL: http://<host>:<listen port>/console/login/LoginForm.jsp.
2. Navigate the domain structure tree to Services/JDBC/Data Sources.
3. Click **New** to create the new data source. Enter the following required information.
   - Name: Select any name for the data source,
   - JNDI name: This field must be set to jdbc/RetailWebServiceDs. The generated code for the service uses this JNDI name to look up the data source.
4. Select the transaction options for the data source. Click **Next**.
5. Enter the database name and user information for the data source. Click **Next.**
6. The connection information for the data source is displayed. Click **Test Configuration** to ensure the connection information is correct. If the information is correct, the following message is displayed: "Connect test succeeded."
7. Click **Next**. Select a server to which to deploy the data source. (This step is not required at this point in the procedure if you want to deploy the data source to a server at a later time.)
8. Click **Finish** to complete the data source setup. The data sources page is displayed, including the new data source.
9. Click the new data source to see the properties page. A default connection pool is created for the data source. Click the **Connection Pool** tab to view the connection pool properties.
10. The generated JDBC connection URL for the data source is displayed in the following format:

    ```
    jdbc:oracle:thin:@<hostname>:<port>:<sid>
    ```

    For example:

    jdbc:oracle:thin:@localhost:1521:orc

11. If the database is a RAC database, the URL should be in the following format:

    ```
    jdbc:oracle:thin:@(DESCRIPTION =(ADDRESS_LIST =(ADDRESS = (PROTOCOL = TCP)(HOST
    ```

```
= <host>)(PORT = <port>))(ADDRESS = (PROTOCOL = TCP)(HOST = <host>)(PORT =
<port>))(LOAD_BALANCE = yes))(CONNECT_DATA =(SERVICE_NAME = <sid>)))
```

For example:

```
jdbc:oracle:thin:@(DESCRIPTION =(ADDRESS_LIST =(ADDRESS = (PROTOCOL = TCP)(HOST
= dbhost1.example.com)(PORT = 1521))(ADDRESS = (PROTOCOL = TCP)(HOST =
dbhost1.example.com)(PORT = 1521))(LOAD_BALANCE = yes))(CONNECT_DATA =(SERVICE_
NAME = orcl)))
```

12. In the Configuration > Connection Pool tab of the data source, set the following properties.
    - Initial capacity: Set the value to 20 connections. This value should be increased or decreased based on the expected load on the server.
    - Maximum capacity: Set the value to 100 connections. This value should be increased or decreased based on the expected load on the server.
    - Capacity Increment: Set the value to 20 connections. This value should be increased or decreased based on the expected load on the server.
    - Inactive Connection Time-out: This property is available in the advanced section of the connection pool configuration. Set the value of this property to 60 seconds.
    - Remove Infected Connections Enabled: This check box must be unchecked.
13. Restart the WebLogic instance to reflect the data source changes.